Arm® A64 Instruction Set Architecture
Armv8, for Armv8-A architecture profile
Arm A64 Instruction Set Architecture
Armv8, for Armv8-A architecture profile

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.

Release Information

For information on the change history and known issues for this release, see the Release Notes in the A64 ISA XML for Armv8.8 (2021-12).

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with "™" or © are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the Arm’s trademark usage guidelines http://www.arm.com/company/policies/trademarks.

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

This release covers multiple versions of the architecture. The content relating to different versions is given different quality ratings.

The information related to the 2021 Architecture Extensions is at Alpha quality. Alpha quality means that most major features of the specification are described in the manual, some features and details might be missing.

The information related to the remaining Armv8-A features which was also published in previous releases, is at Beta quality. Beta quality means that all major features of the specification are described, some details might be missing.
Web Address

http://www.arm.com

Progressive Terminology Commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact terms@arm.com.
A64 -- Base Instructions (alphabetic order)

**ADC**
Add with Carry.

**ADCS**
Add with Carry, setting flags.

**ADD (extended register)**
Add (extended register).

**ADD (immediate)**
Add (immediate).

**ADD (shifted register)**
Add (shifted register).

**ADDG**
Add with Tag.

**ADDS (extended register)**
Add (extended register), setting flags.

**ADDS (immediate)**
Add (immediate), setting flags.

**ADDS (shifted register)**
Add (shifted register), setting flags.

**ADR**
Form PC-relative address.

**ADRP**
Form PC-relative address to 4KB page.

**AND (immediate)**
Bitwise AND (immediate).

**AND (shifted register)**
Bitwise AND (shifted register).

**ANDS (immediate)**
Bitwise AND (immediate), setting flags.

**ANDS (shifted register)**
Bitwise AND (shifted register), setting flags.

**ASR (immediate)**
Arithmetic Shift Right (immediate): an alias of SBFM.

**ASR (register)**
Arithmetic Shift Right (register): an alias of ASRV.

**ASRV**
Arithmetic Shift Right Variable.

**AT**
Address Translate: an alias of SYS.

**AUTDA, AUTDZA**
Authenticate Data address, using key A.

**AUTDB, AUTDZB**
Authenticate Data address, using key B.

**AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA**
Authenticate Instruction address, using key A.

**AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB**
Authenticate Instruction address, using key B.

**AXFLAG**
Convert floating-point condition flags from Arm to external format.

**B**
Branch.

**B.cond**
Branch conditionally.

**BC.cond**
Branch Consistent conditionally.

**BFC**
Bitfield Clear: an alias of BFM.

**BFI**
Bitfield Insert: an alias of BFM.

**BFM**
Bitfield Move.

**BFXIL**
Bitfield extract and insert at low end: an alias of BFM.

**BIC (shifted register)**
Bitwise Bit Clear (shifted register).

**BICS (shifted register)**
Bitwise Bit Clear (shifted register), setting flags.

**BL**
Branch with Link.
BLR: Branch with Link to Register.

BLRAA, BLRAAZ, BLRAB, BLRABZ: Branch with Link to Register, with pointer authentication.

BR: Branch to Register.

BRAA, BRAAZ, BRAB, BRABZ: Branch to Register, with pointer authentication.

BRK: Breakpoint instruction.

BTI: Branch Target Identification.

CAS, CASA, CASAL, CASL: Compare and Swap word or doubleword in memory.

CASB, CASAB, CASALB, CASLB: Compare and Swap byte in memory.

CASH, CASAH, CASALH, CASLH: Compare and Swap halfword in memory.

CASP, CASPA, CASPAL, CASPL: Compare and Swap Pair of words or doublewords in memory.

CBNZ: Compare and Branch on Nonzero.

CBZ: Compare and Branch on Zero.

CCMN (immediate): Conditional Compare Negative (immediate).

CCMN (register): Conditional Compare Negative (register).

CCMP (immediate): Conditional Compare (immediate).

CCMP (register): Conditional Compare (register).

CFINV: Invert Carry Flag.

CFP: Control Flow Prediction Restriction by Context: an alias of SYS.

CINC: Conditional Increment: an alias of CSINC.

CINV: Conditional Invert: an alias of CSINV.

CLREX: Clear Exclusive.

CLS: Count Leading Sign bits.

CLZ: Count Leading Zeros.


CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).


CMP (immediate): Compare (immediate): an alias of SUBS (immediate).


CMPP: Compare with Tag: an alias of SUBPS.

CNEG: Conditional Negate: an alias of CSNEG.

CPP: Cache Prefetch Prediction Restriction by Context: an alias of SYS.

CPYFP, CPYFM, CPYFE: Memory Copy Forward-only.

CPYFPN, CPYFMRN, CPYFERN: Memory Copy Forward-only, reads and writes non-temporal.

CPYFPRT, CPYFMRT, CPYFERT: Memory Copy Forward-only, reads unprivileged.
CPYFPRTN, CPYFMRTN, CPYFERTN: Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal.
CPYFPRTN, CPYFMRTN, CPYFERTN: Memory Copy Forward-only, reads unprivileged and non-temporal.
CPYFPRTWN, CPYFMRTWN, CPYFERTWN: Memory Copy Forward-only, reads unprivileged, writes non-temporal.
CPYFP, CPYM, CPYE: Memory Copy.
CPYPN, CPYMN, CPYEN: Memory Copy, reads and writes non-temporal.
CPYPRN, CPYMRN, CPYERN: Memory Copy, reads and writes non-temporal.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged, reads and writes non-temporal.
CPYPRTRN, CPYMRTRN, CPYERTRN: Memory Copy, reads unprivileged and non-temporal.
CPYPRRTWN, CPYMRRTWN, CPYERTWN: Memory Copy, reads unprivileged, writes non-temporal.
CPYPT, CPYMT, CPYET: Memory Copy, reads and writes unprivileged.
CPYPTN, CPYMTN, CPYETN: Memory Copy, reads and writes unprivileged and non-temporal.
CPYP, CPYM, CPYE: Memory Copy.
CPYPN, CPYMN, CPYEN: Memory Copy, reads and writes non-temporal.
CPYPRN, CPYMRN, CPYERN: Memory Copy, reads and writes non-temporal.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged, reads and writes non-temporal.
CPYPRTRN, CPYMRTRN, CPYERTRN: Memory Copy, reads unprivileged and non-temporal.
CPYPRRTWN, CPYMRRTWN, CPYERTWN: Memory Copy, reads unprivileged, writes non-temporal.
CPYPT, CPYMT, CPYET: Memory Copy, reads and writes unprivileged.
CPYPTN, CPYMTN, CPYETN: Memory Copy, reads and writes unprivileged and non-temporal.
CPYP, CPYM, CPYE: Memory Copy.
CPYPN, CPYMN, CPYEN: Memory Copy, reads and writes non-temporal.
CPYPRN, CPYMRN, CPYERN: Memory Copy, reads and writes non-temporal.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged, reads and writes non-temporal.
CPYPRTRN, CPYMRTRN, CPYERTRN: Memory Copy, reads unprivileged and non-temporal.
CPYPRRTWN, CPYMRRTWN, CPYERTWN: Memory Copy, reads unprivileged, writes non-temporal.
CPYPT, CPYMT, CPYET: Memory Copy, reads and writes unprivileged.
CPYPTN, CPYMTN, CPYETN: Memory Copy, reads and writes unprivileged and non-temporal.
CPYP, CPYM, CPYE: Memory Copy.
CPYPN, CPYMN, CPYEN: Memory Copy, reads and writes non-temporal.
CPYPRN, CPYMRN, CPYERN: Memory Copy, reads and writes non-temporal.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged, reads and writes non-temporal.
CPYPRTRN, CPYMRTRN, CPYERTRN: Memory Copy, reads unprivileged and non-temporal.
CPYPRRTWN, CPYMRRTWN, CPYERTWN: Memory Copy, reads unprivileged, writes non-temporal.
CPYPT, CPYMT, CPYET: Memory Copy, reads and writes unprivileged.
CPYPTN, CPYMTN, CPYETN: Memory Copy, reads and writes unprivileged and non-temporal.
CPYP, CPYM, CPYE: Memory Copy.
CPYPN, CPYMN, CPYEN: Memory Copy, reads and writes non-temporal.
CPYPRN, CPYMRN, CPYERN: Memory Copy, reads and writes non-temporal.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged.
CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged, reads and writes non-temporal.
CPYPRTRN, CPYMRTRN, CPYERTRN: Memory Copy, reads unprivileged and non-temporal.
CPYPRRTWN, CPYMRRTWN, CPYERTWN: Memory Copy, reads unprivileged, writes non-temporal.
CRC32CB, CRC32CH, CRC32CW, CRC32CX: CRC32C checksum.
CSDB: Consumption of Speculative Data Barrier.
CSEL: Conditional Select.
CSET: Conditional Set: an alias of CSINC.
CSETM: Conditional Set Mask: an alias of CSINV.
CSINC: Conditional Select Increment.
CSINV: Conditional Select Invert.
CSNEG: Conditional Select Negation.
DC: Data Cache operation: an alias of SYS.
DCPS1: Debug Change PE State to EL1..
DCPS2: Debug Change PE State to EL2..
DCPS3: Debug Change PE State to EL3.
DGH: Data Gathering Hint.
DMB: Data Memory Barrier.
DRPS: Debug restore process state.
DSB: Data Synchronization Barrier.
DVP: Data Value Prediction Restriction by Context: an alias of SYS.
EON (shifted register): Bitwise Exclusive OR NOT (shifted register).
EOR (immediate): Bitwise Exclusive OR (immediate).
EOR (shifted register): Bitwise Exclusive OR (shifted register).
ERET: Exception Return.
ERETAA, ERETAB: Exception Return, with pointer authentication.
ESR: Error Synchronization Barrier.
EXTR: Extract register.
GMI: Tag Mask Insert.
HINT: Hint instruction.
HLT: Halt instruction.
HVC: Hypervisor Call.
IC: Instruction Cache operation: an alias of SYS.
IRG: Insert Random Tag.
ISB: Instruction Synchronization Barrier.
LD64B: Single-copy Atomic 64-byte Load.
LDADD, LDADDA, LDADDAL, LDADDL: Atomic add on word or doubleword in memory.
LDADDB, LDADDAB, LDADDALB, LDADDLB: Atomic add on byte in memory.
LDADDH, LDADDAH, LDADDALH, LDADDLH: Atomic add on halfword in memory.
LDAPR: Load-Acquire RCpc Register.
LDAPRB: Load-Acquire RCpc Register Byte.
LDAPRH: Load-Acquire RCpc Register Halfword.
LDAPUR: Load-Acquire RCpc Register (unscaled).
LDAPURB: Load-Acquire RCpc Register Byte (unscaled).
LDAPURH: Load-Acquire RCpc Register Halfword (unscaled).
LDAPURSB: Load-Acquire RCpc Register Signed Byte (unscaled).
LDAPURSH: Load-Acquire RCpc Register Signed Halfword (unscaled).
LDAPURSW: Load-Acquire RCpc Register Signed Word (unscaled).
LDAR: Load-Acquire Register.
LDARB: Load-Acquire Register Byte.
LDARH: Load-Acquire Register Halfword.
LDAXP: Load-Acquire Exclusive Pair of Registers.
LDAXR: Load-Acquire Exclusive Register.
LDAXRB: Load-Acquire Exclusive Register Byte.
LDAXRH: Load-Acquire Exclusive Register Halfword.
LDCLR, LDCLRA, LDCLRAL, LDCLRL: Atomic bit clear on word or doubleword in memory.
LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB: Atomic bit clear on byte in memory.
LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH: Atomic bit clear on halfword in memory.
LDEOR, LDEORA, LDEORAL, LDEORL: Atomic exclusive OR on word or doubleword in memory.
LDEORB, LDEORAB, LDEORALB, LDEORLB: Atomic exclusive OR on byte in memory.
LDEORH, LDEORAH, LDEORALH, LDEORLH: Atomic exclusive OR on halfword in memory.
LDG: Load Allocation Tag.
LDGM: Load Tag Multiple.
LDLAR: Load LOAcquire Register.
LDLARB: Load LOAcquire Register Byte.
LDLARH: Load LOAcquire Register Halfword.
LDNP: Load Pair of Registers, with non-temporal hint.
LDP: Load Pair of Registers.
LDPSW: Load Pair of Registers Signed Word.
LDR (immediate): Load Register (immediate).
LDR (literal): Load Register (literal).
LDR (register): Load Register (register).
LDRAA, LDRAA: Load Register, with pointer authentication.
LDRB (immediate): Load Register Byte (immediate).
LDRB (register): Load Register Byte (register).
LDRH (immediate): Load Register Halfword (immediate).
LDRH (register): Load Register Halfword (register).
LDRSB (immediate): Load Register Signed Byte (immediate).
LDRSB (register): Load Register Signed Byte (register).
LDRSH (immediate): Load Register Signed Halfword (immediate).
LDRSH (register): Load Register Signed Halfword (register).
LDRSW (immediate): Load Register Signed Word (immediate).
LDRSW (literal): Load Register Signed Word (literal).
LDRSW (register): Load Register Signed Word (register).
LDSET, LDSETA, LDSETAL, LDSETL: Atomic bit set on word or doubleword in memory.
LDSETB, LDSETAB, LDSETALB, LDSETLB: Atomic bit set on byte in memory.
LDSETH, LDSETAH, LDSETALH, LDSETLH: Atomic bit set on halfword in memory.
LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL: Atomic signed maximum on word or doubleword in memory.
LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB: Atomic signed maximum on byte in memory.
LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH: Atomic signed maximum on halfword in memory.
LDSMIN, LDSMINA, LDSMINAL, LDSMINL: Atomic signed minimum on word or doubleword in memory.
LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB: Atomic signed minimum on byte in memory.
LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH: Atomic signed minimum on halfword in memory.
LDTR: Load Register (unprivileged).
LDTRB: Load Register Byte (unprivileged).
LDTRH: Load Register Halfword (unprivileged).
LDTRSB: Load Register Signed Byte (unprivileged).
LDTRSH: Load Register Signed Halfword (unprivileged).
LDTRSW: Load Register Signed Word (unprivileged).
LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL: Atomic unsigned maximum on word or doubleword in memory.
LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB: Atomic unsigned maximum on byte in memory.
LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH: Atomic unsigned maximum on halfword in memory.
LDumin, LDuminA, LDuminAL, LDuminL: Atomic unsigned minimum on word or doubleword in memory.
LDuminB, LDuminAB, LDuminALB, LDuminLB: Atomic unsigned minimum on byte in memory.
LDuminH, LDuminAH, LDuminALH, LDuminLH: Atomic unsigned minimum on halfword in memory.
LDUR: Load Register (unscaled).
LDURB: Load Register Byte (unscaled).
LDURH: Load Register Halfword (unscaled).
LDURSB: Load Register Signed Byte (unscaled).
LDURSH: Load Register Signed Halfword (unscaled).
LDURSW: Load Register Signed Word (unscaled).
LDXP: Load Exclusive Pair of Registers.
LDXR: Load Exclusive Register.
LDXRB: Load Exclusive Register Byte.
LDXRH: Load Exclusive Register Halfword.
LSI (immediate): Logical Shift Left (immediate): an alias of UBFM.
LSI (register): Logical Shift Left (register): an alias of LSLV.
LSLV: Logical Shift Left Variable.
LSR (immediate): Logical Shift Right (immediate): an alias of UBFM.

LSR (register): Logical Shift Right (register): an alias of LSRV.

LSRV: Logical Shift Right Variable.

MADD: Multiply-Add.

MNEG: Multiply-Negate: an alias of MSUB.

MOV (bitmask immediate): Move (bitmask immediate): an alias of ORR (immediate).

MOV (inverted wide immediate): Move (inverted wide immediate): an alias of MOVN.


MOV (to/from SP): Move between register and stack pointer: an alias of ADD (immediate).

MOV (wide immediate): Move (wide immediate): an alias of MOVZ.

MOVK: Move wide with keep.

MOVN: Move wide with NOT.

MOVZ: Move wide with zero.

MRS: Move System Register.

MSR (immediate): Move immediate value to Special Register.

MSR (register): Move general-purpose register to System Register.

MSUB: Multiply-Subtract.

MUL: Multiply: an alias of MADD.

MVN: Bitwise NOT: an alias of ORN (shifted register).


NEGS: Negate, setting flags: an alias of SUBS (shifted register).

NGC: Negate with Carry: an alias of SBC.

NGCS: Negate with Carry, setting flags: an alias of SBCS.

NOP: No Operation.

ORN (shifted register): Bitwise OR NOT (shifted register).

ORR (immediate): Bitwise OR (immediate).

ORR (shifted register): Bitwise OR (shifted register).

PACDA, PACDZA: Pointer Authentication Code for Data address, using key A.

PACDB, PACDZB: Pointer Authentication Code for Data address, using key B.

PACGA: Pointer Authentication Code, using Generic key.

PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA: Pointer Authentication Code for Instruction address, using key A.

PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB: Pointer Authentication Code for Instruction address, using key B.

PRFM (immediate): Prefetch Memory (immediate).

PRFM (literal): Prefetch Memory (literal).

PRFM (register): Prefetch Memory (register).

PRFUM: Prefetch Memory (unscaled offset).
PSB CSYNC: Profiling Synchronization Barrier.

PSSBB: Physical Speculative Store Bypass Barrier: an alias of DSB.

RBIT: Reverse Bits.

RET: Return from subroutine.

RETA, RETB: Return from subroutine, with pointer authentication.

REV: Reverse Bytes.

REV16: Reverse bytes in 16-bit halfwords.

REV32: Reverse bytes in 32-bit words.

REV64: Reverse Bytes: an alias of REV.

RMIF: Rotate, Mask Insert Flags.

ROR (immediate): Rotate right (immediate): an alias of EXTR.

ROR (register): Rotate Right (register): an alias of RORV.

RORV: Rotate Right Variable.

SB: Speculation Barrier.

SBC: Subtract with Carry.

SRCS: Subtract with Carry, setting flags.

SBFIZ: Signed Bitfield Insert in Zero: an alias of SBFM.

SBFM: Signed Bitfield Move.

SBFX: Signed Bitfield Extract: an alias of SBFM.

SDIV: Signed Divide.

SETF8, SETF16: Evaluation of 8 or 16 bit flag values.

SETGP, SETGM, SETGE: Memory Set with tag setting.

SETGPN, SETGMN, SETGEN: Memory Set with tag setting, non-temporal.

SETGPT, SETGMT, SETGET: Memory Set with tag setting, unprivileged.

SETGPTN, SETGMTN, SETGETN: Memory Set with tag setting, unprivileged and non-temporal.

SETP, SETM, SETE: Memory Set.

SETPN, SETMN, SETEN: Memory Set, non-temporal.

SETPT, SETMT, SETET: Memory Set, unprivileged.

SETPTN, SETMTN, SETETN: Memory Set, unprivileged and non-temporal.

SEV: Send Event.

SEVL: Send Event Local.

SMADDL: Signed Multiply-Add Long.

SMC: Secure Monitor Call.

SMNEGL: Signed Multiply-Negate Long: an alias of SMSUBL.

SMSUBL: Signed Multiply-Subtract Long.

SMUL: Signed Multiply High.
**SMULL**: Signed Multiply Long: an alias of SMADDL.

**SSBB**: Speculative Store Bypass Barrier: an alias of DSB.

**ST2G**: Store Allocation Tags.

**ST64B**: Single-copy Atomic 64-byte Store without Return.

**ST64BV**: Single-copy Atomic 64-byte Store with Return.

**ST64BV0**: Single-copy Atomic 64-byte EL0 Store with Return.

**STADD, STADDL**: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDL.

**STADDB, STADDLB**: Atomic add on byte in memory, without return: an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB.

**STADDDH, STADDDLH**: Atomic add on halfword in memory, without return: an alias of LDADDDH, LDADDDAH, LDADDDLH, LDADDLH.

**STCLR, STCLRL**: Atomic bit clear on word or doubleword in memory, without return: an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL.

**STCLRB, STCLRLB**: Atomic bit clear on byte in memory, without return: an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.

**STCLRH, STCLRLH**: Atomic bit clear on halfword in memory, without return: an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.

**STEO, STEORL**: Atomic exclusive OR on word or doubleword in memory, without return: an alias of LDEOR, LDEORA, LDEORAL, LDEORL.

**STEOB, STEORLB**: Atomic exclusive OR on byte in memory, without return: an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB.

**STEORH, STEORLH**: Atomic exclusive OR on halfword in memory, without return: an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH.

**STG**: Store Allocation Tag.

**STGM**: Store Tag Multiple.

**STGP**: Store Allocation Tag and Pair of registers.

**STLLR**: Store LORelease Register.

**STLLRB**: Store LORelease Register Byte.

**STLLRH**: Store LORelease Register Halfword.

**STLR**: Store-Release Register.

**STLRB**: Store-Release Register Byte.

**STLRH**: Store-Release Register Halfword.

**STLUR**: Store-Release Register (unscaled).

**STLURB**: Store-Release Register Byte (unscaled).

**STLURH**: Store-Release Register Halfword (unscaled).

**STLXP**: Store-Release Exclusive Pair of registers.

**STLXR**: Store-Release Exclusive Register.

**STLXRB**: Store-Release Exclusive Register Byte.

**STLXRH**: Store-Release Exclusive Register Halfword.

**STNP**: Store Pair of Registers, with non-temporal hint.
**STP**: Store Pair of Registers.

**STR (immediate)**: Store Register (immediate).

**STR (register)**: Store Register (register).

**STRB (immediate)**: Store Register Byte (immediate).

**STRB (register)**: Store Register Byte (register).

**STRH (immediate)**: Store Register Halfword (immediate).

**STRH (register)**: Store Register Halfword (register).

**STSET, STSETL**: Atomic bit set on word or doubleword in memory, without return: an alias of LDSET, LDSETA, LDSETAL, LDSETL.

**STSETB, STSETLB**: Atomic bit set on byte in memory, without return: an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB.

**STSETH, STSETLH**: Atomic bit set on halfword in memory, without return: an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH.

**STSMAX, STSMAXL**: Atomic signed maximum on word or doubleword in memory, without return: an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.

**STSMAXB, STSMAXLB**: Atomic signed maximum on byte in memory, without return: an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.

**STSMAXH, STSMAXLH**: Atomic signed maximum on halfword in memory, without return: an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.

**STSMIN, STSMINL**: Atomic signed minimum on word or doubleword in memory, without return: an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.

**STSMINB, STSMINLB**: Atomic signed minimum on byte in memory, without return: an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.

**STSMINH, STSMINLH**: Atomic signed minimum on halfword in memory, without return: an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.

**STTR**: Store Register (unprivileged).

**STTRB**: Store Register Byte (unprivileged).

**STTRH**: Store Register Halfword (unprivileged).

**STUMAX, STUMAXL**: Atomic unsigned maximum on word or doubleword in memory, without return: an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.

**STUMAXB, STUMAXLB**: Atomic unsigned maximum on byte in memory, without return: an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.

**STUMAXH, STUMAXLH**: Atomic unsigned maximum on halfword in memory, without return: an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.

**STUMIN, STUMINL**: Atomic unsigned minimum on word or doubleword in memory, without return: an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.

**STUMINB, STUMINLB**: Atomic unsigned minimum on byte in memory, without return: an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.

**STUMINH, STUMINLH**: Atomic unsigned minimum on halfword in memory, without return: an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.

**STUR**: Store Register (unscaled).

**STURB**: Store Register Byte (unscaled).

**STURH**: Store Register Halfword (unscaled).

**STXP**: Store Exclusive Pair of registers.
**STXR**: Store Exclusive Register.

**STXRB**: Store Exclusive Register Byte.

**STXRH**: Store Exclusive Register Halfword.

**STZ2G**: Store Allocation Tags, Zeroing.

**STZG**: Store Allocation Tag, Zeroing.

**STZGM**: Store Tag and Zero Multiple.

**SUB (extended register)**: Subtract (extended register).

**SUB (immediate)**: Subtract (immediate).

**SUB (shifted register)**: Subtract (shifted register).

**SUBG**: Subtract with Tag.

**SUBP**: Subtract Pointer.

**SUBPS**: Subtract Pointer, setting Flags.

**SUBS (extended register)**: Subtract (extended register), setting flags.

**SUBS (immediate)**: Subtract (immediate), setting flags.

**SUBS (shifted register)**: Subtract (shifted register), setting flags.

**SVC**: Supervisor Call.

**SWP, SWPA, SWPAL, SWPL**: Swap word or doubleword in memory.

**SWPB, SWPAB, SWPALB, SWPLB**: Swap byte in memory.

**SWPH, SWPAH, SWPALH, SWPLH**: Swap halfword in memory.

**SXTB**: Signed Extend Byte: an alias of SBFM.

**SXTH**: Sign Extend Halfword: an alias of SBFM.

**SXTW**: Sign Extend Word: an alias of SBFM.

**SYS**: System instruction.

**SYSL**: System instruction with result.

**TBNZ**: Test bit and Branch if Nonzero.

**TBZ**: Test bit and Branch if Zero.

**TLBI**: TLB Invalidate operation: an alias of SYS.

**TSB CSYNC**: Trace Synchronization Barrier.

**TST (immediate)**: Test bits (immediate): an alias of ANDS (immediate).

**TST (shifted register)**: Test (shifted register): an alias of ANDS (shifted register).

**UBFIZ**: Unsigned Bitfield Insert in Zero: an alias of UBFM.

**UBFM**: Unsigned Bitfield Move.

**UBFX**: Unsigned Bitfield Extract: an alias of UBFM.

**UDF**: Permanently Undefined.

**UDIV**: Unsigned Divide.

**UMADDL**: Unsigned Multiply-Add Long.
UMNEGL: Unsigned Multiply-Negate Long: an alias of UMSUBL.

UMSUBL: Unsigned Multiply-Subtract Long.

UMULH: Unsigned Multiply High.

UMULL: Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.

WFE: Wait For Event.

WFET: Wait For Event with Timeout.

WFI: Wait For Interrupt.

WFIT: Wait For Interrupt with Timeout.

XAFLAG: Convert floating-point condition flags from external format to Arm format.

XPACD, XPACI, XPACLRI: Strip Pointer Authentication Code.

YIELD: YIELD.
ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

op S

32-bit (sf == 0)

ADC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ADC <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rd</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

op S

32-bit (sf == 0)

ADCS <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ADCS <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

32-bit (sf == 0)

ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit (sf == 1)

ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
<R> Is a width specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>
If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

```plaintext
bits(datasize) result;
bias(datasize) operand1 = if n == 31 then SP[] else X[n];
bias(datasize) operand2 = ExtendReg(m, extend_type, shift);
(result, -) = AddWithCarry(operand1, operand2, '0');
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. This instruction is used by the alias MOV (to/from SP).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| sf | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | sh | imm12 | Rn | Rd |

op S

**32-bit (sf == 0)**

ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}

**64-bit (sf == 1)**

ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12; Zeros(12), datasize);
```

### Assembler Symbols

- `<Wd|WSP>` is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to/from SP)</td>
<td>sh == '0' &amp; imm12 == '000000000000' &amp; (Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
(result, -) = AddWithCarry(operand1, operand2, '0');
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

**Integer (FEAT_MTE)**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op3 | uimm6 | (0) | (0) | uimm4 | Xn | Xd |

**ADDG** `<Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>`

```plaintext
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

**Assembler Symbols**

- `<Xd|SP>`: Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- `<Xn|SP>`: Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<uimm6>`: Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
- `<uimm4>`: Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

**Operation**

```plaintext
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';
(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>imm3</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit (sf == 1)

ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>
If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (immediate)

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADD <Wd>, <Wn|WSP>, #<imm>{, <shift>}

64-bit (sf == 1)

ADD <Xd>, <Xn|SP>, #<imm>{, <shift>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

bits(datasize) imm;

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register).

### 32-bit (sf == 0)

ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit (sf == 1)

ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

### Assembler Symbols

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<shift>** Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<amount>** For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

```c
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

```
0 | immlo | 1 | 0 | 0 | 0 | 0 | immhi |
   op

ADR <Xd>, <label>

integer d = UInt(Rd);
bits(64) imm;
imm = SignExtend(immhi:immlo, 64);
```

**Assembler Symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<label>` Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

**Operation**

```
bits(64) base = PC[];
Xd[d] = base + imm;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.

\[
\begin{array}{cccccccccccccc}
\hline
\text{immlo} & 1 & 0 & 0 & 0 & 0 & \text{immhi} & \text{Rd} \\
\end{array}
\]

**Asmplers Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<label>** Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

**Operation**

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd) \\
\text{bits(64)} &\text{ imm;} \\
\text{imm} &= \text{SignExtend}(\text{immhi:immlo}:\text{Zeros}(12), 64); \\
\text{bits(64) } &\text{base} = \text{PC}[]; \\
\text{base}<11:0> &= \text{Zeros}(12); \\
X[d] &= \text{base} + \text{imm};
\end{align*}
\]
AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 0 0 1 0 0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 & N == 0)

AND <Wd|WSP>, <Wn>, #<imm>

64-bit (sf == 1)

AND <Xd|SP>, <Xn>, #<imm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
result = operand1 AND imm;
if d == 31 then
  SP[] = result;
else
  X[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
AND (shifted register)

Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 0 0 1 0 1 0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(datasize) result;
result = operand1 AND operand2;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ANDS (immediate)

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate).

32-bit ($sf == 0 \&\& N == 0$)

ANDS $<Wd>$, $<Wn>$, $\#<imm>$

64-bit ($sf == 1$)

ANDS $<Xd>$, $<Xn>$, $\#<imm>$

integer $d = UInt(Rd);$  
integer $n = UInt(Rn);$  
integer datasize = if $sf == '1$' then 64 else 32;

bits(datasize) imm;  
if $sf == '0' \&\& N != '0$' then UNDEFINED;  
(imm, -) = DecodeBitMasks(N, immr, imms, TRUE);

Assembler Symbols

$<Wd>$  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
$<Wn>$  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
$<Xd>$  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
$<Xn>$  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
$<imm>$ For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
          For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (immediate)</td>
<td>$Rd == '11111'$</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;  
bits(datasize) operand1 = $X[n]$;  
result = operand1 AND imm;  
PSTATE.$<N,Z,C,V>$ = result<datasize-1>:IsZeroBit(result):'00';  
$X[d]$ = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register).

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(datasize) result;

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

| sf | 0 0 0 1 0 0 1 1 | N | immr | x | 1 | 1 | 1 | 1 | Rn | Rd |
|----|----------------|----|------|---|---|---|---|---|---|----|----|
| opc | imms |

32-bit (sf == 0 && N == 0 && imms == 011111)

ASR <Wd>, <Wn>, #<shift>

is equivalent to

SBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1 && imms == 111111)

ASR <Xd>, <Xn>, #<shift>

is equivalent to

SBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ASR (register)**

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of **ASRV**. This means:

- The encodings in this description are named to match the encodings of **ASRV**.
- The description of **ASRV** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
</table>

### 32-bit (sf == 0)

ASR <Wd>, <Wn>, <Wm>

is equivalent to

ASRV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

### 64-bit (sf == 1)

ASR <Xd>, <Xn>, <Xm>

is equivalent to

ASRV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

**Assembler Symbols**

| <Wd> | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn> | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm> | Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field. |
| <Xd> | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn> | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Xm> | Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field. |

**Operation**

The description of **ASRV** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias `ASR (register)`.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op2</td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ASRV `<Wd>, <Wn>, <Wm>`

64-bit (sf == 1)

ASRV `<Xd>, <Xn>, <Xm>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

```
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AT

Address Translate. For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>L                  CRn                  CRm</td>
</tr>
</tbody>
</table>

AT <at_op>, <Xt>

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>, <Xt>

and is the preferred disassembly when SysOp(op1, '0111', CRm, op2) == Sys_AT.

Assembler Symbols

<at_op> Is an AT instruction name, as listed for the AT system instruction group, encoded in "op1:CRm<0>:op2":

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm&lt;0&gt;</th>
<th>op2</th>
<th>&lt;at_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>000</td>
<td>S1E1R</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>001</td>
<td>S1E1W</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>010</td>
<td>S1E0R</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>011</td>
<td>S1E0W</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>000</td>
<td>S1E1RP</td>
<td>FEAT_PAN2</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>001</td>
<td>S1E1WP</td>
<td>FEAT_PAN2</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>000</td>
<td>S1E2R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>001</td>
<td>S1E2W</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>100</td>
<td>S12E1R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>101</td>
<td>S12E1W</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>110</td>
<td>S12E0R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>111</td>
<td>S12E0W</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>000</td>
<td>S1E3R</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>001</td>
<td>S1E3W</td>
<td>-</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
**AUTDA, AUTDZA**

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A. The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

**Integer**

(FEAT_PAuth)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0|
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 1  | 0  | Rn | Rd |

**AUTDA (Z == 0)**

AUTDA <Xd>, <Xn|SP>

**AUTDZA (Z == 1 && Rn == 11111)**

AUTDZA <Xd>

```plaintext
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
  UNDEFINED;
if Z == '0' then // AUTDA
  if n == 31 then source_is_sp = TRUE;
else // AUTDZA
  if n != 31 then UNDEFINED;
```

**Assembler Symbols**

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
if HavePACExt() then
  if source_is_sp then
    X[d] = AuthDA(X[d], SP[], FALSE);
  else
    X[d] = AuthDA(X[d], X[n], FALSE);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B. The address is in the general-purpose register that is specified by <Xd>.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDB.
- The value zero, for AUTDZB.

If the authentication fails, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Integer
(FEAT_PAuth)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 1  | 1  | Rn | Rd |

AUTDB (Z == 0)

AUTDB <Xd>, <Xn|SP>

AUTDZB (Z == 1 && Rn == 11111)

AUTDZB <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UNDEFINED;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```java
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDB(X[d], SP[], FALSE);
    else
        X[d] = AuthDB(X[d], X[n], FALSE);
```
AUTHIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:
- In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: Integer and System

\[
\begin{array}{cccccccccccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & Z & 1 & 0 & 0 & Rn & Rd \\
\end{array}
\]

AUTHIA \( (Z == 0) \)

\[
\begin{align*}
\text{AUTIA} &\ <Xd>, \ <Xn|SP> \\
\end{align*}
\]

AUTHIZA \( (Z == 1 \ & \ Rn == 1111) \)

AUTIZA \ <Xd>

\[
\begin{align*}
\text{boolean source_is_sp} &\ = \text{FALSE}; \\
\text{integer } d &\ = \text{UInt(Rd)}; \\
\text{integer } n &\ = \text{UInt(Rn)}; \\
\text{if } !\text{HavePACExt}() \text{ then } \\
\text{UNDEFINED;} \\
\text{if } Z == '0' \text{ then // AUTIA} \\
\text{if } n == 31 \text{ then source_is_sp = TRUE;} \\
\text{else // AUTIZA} \\
\text{if } n != 31 \text{ then UNDEFINED;} \\
\end{align*}
\]

\[
\begin{array}{cccccccccccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & x & 1 & 1 & 0 & x & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

System
\( (\text{FEAT_PAuth}) \)
AUTIA1716 (\text{CRm} == 0001 && \text{op2} == 100)

AUTIA1716

AUTIASP (\text{CRm} == 0011 && \text{op2} == 101)

AUTIASP

AUTIAZ (\text{CRm} == 0011 && \text{op2} == 100)

AUTIAZ

integer \(d\);
integer \(n\);
boolean source\_is\_sp = FALSE;

case \text{CRm:op2} of
  when '0011 100'    // AUTIAZ
    \(d = 30\);
    \(n = 31\);
  when '0011 101'    // AUTIASP
    \(d = 30\);
    source\_is\_sp = TRUE;
  when '0001 100'    // AUTIA1716
    \(d = 17\);
    \(n = 16\);
  when '0001 000' SEE "PACIA";
  when '0001 010' SEE "PACIB";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 01x' SEE "PACIB";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";

Assembler Symbols

\text{<Xd>} \quad \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.}

\text{<Xn|SP>} \quad \text{Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.}

Operation

\text{if HavePACExt() then}
  \text{\quad if source\_is\_sp then}
    \(X[d] = \text{AuthIA}(X[d], \text{SP[]}, FALSE)\);
  \text{\quad else}
    \(X[d] = \text{AuthIA}(X[d], X[n], FALSE)\);
AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:
- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: Integer and System

### Integer (FEAT_PAuth)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### AUTIB (Z == 0)

AUTIB <Xd>, <Xn|SP>

### AUTIZB (Z == 1 && Rn == 1111)

AUTIZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIB
    if n == 31 then source_is_sp = TRUE;
else // AUTIZB
    if n != 31 then UNDEFINED;

### System (FEAT_PAuth)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CRm | op2
AUTIB1716 (CRm == 0001 && op2 == 110)

AUTIB1716

AUTIBSP (CRm == 0011 && op2 == 111)

AUTIBSP

AUTIBZ (CRm == 0011 && op2 == 110)

AUTIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 110'    // AUTIBZ
        d = 30;
        n = 31;
    when '0011 111'    // AUTIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 110'    // AUTIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
    if source_is_sp then
        X[d] = AuthIB(X[d], SP[], FALSE);
    else
        X[d] = AuthIB(X[d], X[n], FALSE);

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AXFLAG

Convert floating-point condition flags from Arm to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

**System**

(_FEAT_FlagM2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | (0)| (0)| (0)| (0)| 0  | 1  | 0  | 1  | 1  | 1  | 1  |

**AXFLAG**

if !HaveFlagFormatExt() then UNDEFINED;

**Operation**

bit Z = PSTATE.Z OR PSTATE.V;
bit C = PSTATE.C AND NOT(PSTATE.V);

PSTATE.N = '0';
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = '0';

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{imm26} \]

\[ \text{op} \]

\[ B \ <\text{label}> \]

bits(64) offset = \text{SignExtend}(\text{imm26}:\'00\', 64);

\[ \text{Assembler Symbols} \]

\[ <\text{label}> \]

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

\[ \text{Operation} \]

\[ \text{BranchTo} (\text{PC}[] + \text{offset}, \text{BranchType_DIR}, \text{FALSE}); \]

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
### B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | imm19 | 0 | cond |
|---|---|---|
| 0 1 0 1 0 1 0 | 0 |

\[
\text{B.<cond> <label>}
\]

bits(64) offset = \texttt{SignExtend}(imm19:'00', 64);

### Assembler Symbols

- `<cond>` is one of the standard conditions, encoded in the "cond" field in the standard way.
- `<label>` is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

### Operation

\[
\text{if } \texttt{ConditionHolds}(\text{cond}) \text{ then } \text{BranchTo}(\text{PC}[1] + \text{offset}, \text{BranchType_DIR}, \text{TRUE});
\]

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BC.cond

Branch Consistent conditionally to a label at a PC-relative offset, with a hint that this branch will behave very consistently and is very unlikely to change direction.

19-bit signed PC-relative branch offset
(FEAT_HBC)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | imm19 | 1  | cond |

BC.<cond> <label>

if !HaveFeatHBC() then UNDEFINED;
bits(64) offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.
<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

if ConditionHolds(cond) then
  BranchTo(PC[] + offset, BranchType_DIR, TRUE);

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFC

Bitfield Clear sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other
destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

Leaving other bits unchanged
(FEAT_ASMv8p2)

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

BFC <Wd>, #<lsb>, #<width>
is equivalent to

BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 && N == 1)

BFC <Xd>, #<lsb>, #<width>
is equivalent to

BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BFI

Bitfield Insert copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

• The encodings in this description are named to match the encodings of BFM.
• The description of BFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>! = 1111</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 & N == 0)

BFI <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 & N == 1)

BFI <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If `imms` is greater than or equal to `immr`, this copies a bitfield of \((imms - immr + 1)\) bits starting from bit position `immr` in the source register to the least significant bits of the destination register. If `imms` is less than `immr`, this copies a bitfield of \((imms + 1)\) bits from the least significant bits of the source register to bit position \(\text{regsize} - \text{immr}\) of the destination register, where `regsize` is the destination register size of 32 or 64 bits. In both cases the other bits of the destination register remain unchanged.

This instruction is used by the aliases `BFC`, `BFI`, and `BFXIL`.

### 32-bit \((sf == 0 \&\& N == 0)\)

```plaintext
BFM <Wd>, <Wn>, #<immr>, #<imms>
```

### 64-bit \((sf == 1 \&\& N == 1)\)

```plaintext
BFM <Xd>, <Xn>, #<immr>, #<imms>
```

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer datasize = if \(sf == '1'\) then 64 else 32;

integer \(R\);
bits(datasize) \(wmask\);
bits(datasize) \(tmask\);

if \(sf == '1' \&\& N != '1'\) then UNDEFINED;
if \(sf == '0' \&\& (N != '0' || immr<5> != '0' || imms<5> != '0')\) then UNDEFINED;

\(R = \text{UInt}(immr)\);
\((wmask, tmask) = \text{DecodeBitMasks}(N, imms, immr, FALSE)\);

### Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.
- `<Xn>` is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.
- `<immr>` is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.
- `<imms>` is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.
- `<immr>` for the 32-bit variant is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.
- `<imms>` for the 32-bit variant is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BFC</code></td>
<td>(Rn == '11111' &amp;&amp; \text{UInt}(imms) &lt; \text{UInt}(immr))</td>
</tr>
<tr>
<td><code>BFI</code></td>
<td>(Rn != '11111' &amp;&amp; \text{UInt}(imms) &lt; \text{UInt}(immr))</td>
</tr>
<tr>
<td><code>BFXIL</code></td>
<td>(\text{UInt}(imms) &gt;= \text{UInt}(immr))</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) dst = X[d];
bias(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// combine extension bits and result bits
X[d] = (dst AND NOT(tmask)) OR (bot AND tmask);

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**BFXIL**

Bitfield Extract and Insert Low copies a bitfield of `<width>` bits starting from bit position `<lsb>` in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of **BFM**. This means:

- The encodings in this description are named to match the encodings of **BFM**.
- The description of **BFM** gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 1  | 1  | 0  | 0  | 1  | 0  | N  | immr| imms| Rn | Rd |

**32-bit (sf == 0 & N == 0)**

BFXIL `<Wd>`, `<Wn>`, `<lsb>`, `<width>`

is equivalent to

BFM `<Wd>`, `<Wn>`, `<lsb>`, `{<lsb> + <width> - 1}`

and is the preferred disassembly when UInt(imms) >= UInt(immr).

**64-bit (sf == 1 & N == 1)**

BFXIL `<Xd>`, `<Xn>`, `<lsb>`, `<width>`

is equivalent to

BFM `<Xd>`, `<Xn>`, `<lsb>`, `{<lsb> + <width> - 1}`

and is the preferred disassembly when UInt(imms) >= UInt(immr).

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<lsb>` For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- `<width>` For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of **BFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
## BIC (shifted register)

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

BIC `<Wd>`, `<Wn>`, `<Wm>{, <shift> #<amount>}`

### 64-bit (sf == 1)

BIC `<Xd>`, `<Xn>`, `<Xm>{, <shift> #<amount>}`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Operation

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 AND operand2;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 1 1 0 1 0 1 0 shift 1 Rm imm6 Rn Rd
```

**32-bit (sf == 0)**

BICS `<Wd>, <Wn>, <Wm>{, <shift> #<amount>}`

**64-bit (sf == 1)**

BICS `<Xd>, <Xn>, <Xm>{, <shift> #<amount>}`

```
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = Uint(imm6);
```

**Assembler Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
Operation

\[
\text{bits}(\text{datasize}) \text{ operand1} = X[n];
\]
\[
\text{bits}(\text{datasize}) \text{ operand2} = \text{ShiftReg}(m, \text{shift_type}, \text{shift_amount});
\]
\[
\text{bits}(\text{datasize}) \text{ result};
\]
\[
\text{operand2} = \text{NOT}(%s); \quad \text{result} = \text{operand1 AND operand2};
\]
\[
\text{PSTATE.<N,Z,C,V>} = \text{result<datasize-1>:IsZeroBit(result)':'00'};
\]
\[
X[d] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

![Binary representation](image)

**BL**

\[
\begin{array}{ccccccccccccccccccccccc}
1 & 0 & 0 & 1 & 0 & 1 & \text{imm26}
\end{array}
\]

\text{op}

**BL <label>**

bits(64) offset = \text{SignExtend}(\text{imm26}:`00', 64);

**Assembler Symbols**

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

**Operation**

\[
X[30] = \text{PC}[] + 4;
\]

\text{BranchTo} (\text{PC}[] + \text{offset}, \text{BranchType_DIRCALL}, \text{FALSE});
BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 Rn 0 0 0 0 0 Rm |
| Z   op  A  M   | Rn  | Rm     |

BLR <Xn>

integer n = UINT(Rn);

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n];

X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```
BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer
(FEAT_PAuth)

```plaintext
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Key A, zero modifier (Z == 0 & M == 0 & Rm == 11111)

BLRAAZ <Xn>

Key A, register modifier (Z == 1 & M == 0)

BLRAA <Xn>, <Xm|SP>

Key B, zero modifier (Z == 0 & M == 1 & Rm == 11111)

BLRABZ <Xn>

Key B, register modifier (Z == 1 & M == 1)

BLRAB <Xn>, <Xm|SP>

```plaintext
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') & (m == 31));

if !HavePACExt() then
    UNDEFINED;

if Z == '0' & m != 31 then
    UNDEFINED;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.
Operation

\[
\text{bits(64) target} = X[n];
\]

\[
\text{bits(64) modifier} = \begin{cases} \text{SP[]} & \text{if source\_is\_sp} \\ X[m] & \text{else} \end{cases};
\]

\[
\text{if use\_key\_a then} \\
\quad \text{target} = \text{AuthIA}(\text{target, modifier, TRUE}); \\
\text{else} \\
\quad \text{target} = \text{AuthIB}(\text{target, modifier, TRUE});
\]

\[
X[30] = PC[] + 4;
\]

// Value in BTypeNext will be used to set PSTATE.BTYPE

\[
\text{BTypeNext} = '10'; \\
\text{BranchTo}(\text{target, BranchType\_INDCALL}, \text{FALSE});
\]
Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

```
1 1 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

**BR <Xn>**

```plaintext
integer n = UInt(Rn);
```

**Assembler Symbols**

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) target = X[n];

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by &lt;Xn&gt;, using a modifier and the specified key, and branches to the authenticated address. The modifier is:

- In the general-purpose register or stack pointer that is specified by &lt;Xm|SP&gt; for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

**Integer**

(FEAT_PAuth)

```
<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Key A, zero modifier (Z == 0 &amp; M == 0 &amp; Rm == 11111)

```
BRAAZ &lt;Xn&gt;
```

Key A, register modifier (Z == 1 &amp; M == 0)

```
BRAA &lt;Xn&gt;, &lt;Xm|SP&gt;
```

Key B, zero modifier (Z == 0 &amp; M == 1 &amp; Rm == 11111)

```
BRABZ &lt;Xn&gt;
```

Key B, register modifier (Z == 1 &amp; M == 1)

```
BRAB &lt;Xn&gt;, &lt;Xm|SP&gt;
```

```c
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') &amp; (m == 31));
```

```c
if !HavePACExt() then
    UNDEFINED;
```

```c
if Z == '0' &amp; m != 31 then
    UNDEFINED;
```

Assembler Symbols

- &lt;Xn&gt; Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
- &lt;Xm|SP&gt; Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.
Operation

bits(64) target = X[n];

bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
  target = AuthIA(target, modifier, TRUE);
else
  target = AuthIB(target, modifier, TRUE);

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
  if n == 16 || n == 17 then
    BTypeNext = '01';
  else
    BTypeNext = '11';
else
  BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
Breakpoint instruction. A BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value 0x3c, and captures the value of the immediate argument in ESR_ELx.ISS.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------|-----------------------------------------------|
| imm16                          | 0 0 0 0                                       |
```

BRK #<imm>

```c
if HaveBTEExt() then
  SetBTypeCompatible(TRUE);
```

**Assembler Symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

**Operation**

AArch64.SoftwareBreakpoint(imm16);
**BTI**

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions which are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region while \textit{PSTATE.BTYPE} \neq 0b00, a BTI instruction compatible with the current value of \textit{PSTATE.BTYPE} will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand <targets> passed to a BTI instruction determines the values of \textit{PSTATE.BTYPE} which the BTI instruction is compatible with.

**Note**

Within a guarded memory region, when \textit{PSTATE.BTYPE} \neq 0b00, all instructions will generate a Branch Target Exception, other than BRK, BTI, HLT, PACIASP, and PACIBSP, which might not. See the individual instructions for more information.

**System (FEAT_BTI)**

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 1 0 0 x x 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>

**BTI \{<targets>\}**

```assembly
SystemHintOp op;
if CRm:op2 == '0100 xx0' then
  op = SystemHintOp_BTI;
  // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
  SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else
  EndOfInstruction();
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>&lt;targets&gt;</th>
<th>Is the type of indirection, encoded in &quot;op2&lt;2:1&gt;&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{op2&lt;2:1&gt;}</td>
<td>\text{&lt;targets&gt;}</td>
</tr>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>c</td>
</tr>
<tr>
<td>10</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>jc</td>
</tr>
</tbody>
</table>
case op of
  when SystemHintOp_YIELD
      Hint_Yield();
  when SystemHintOp_DGH
      Hint_DGH();
  when SystemHintOp_WFE
      Hint_WFE(1, WFxType_WFE);
  when SystemHintOp_WFI
      Hint_WFI(1, WFxType_WFI);
  when SystemHintOp_SEV
      SendEvent();
  when SystemHintOp_SEVL
      SendEventLocal();
  when SystemHintOp_ESB
      SynchronizeErrors();
      AArch64.ESBOperation();
      if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
      ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
      TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
      ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
      SetBTypeNext('00');
  otherwise    // do nothing
CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

**No offset**

*(FEAT_LSE)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 1  | L  | 1  | Rs | 00 | 1  | 1  | 1  | 1  | 1  |  Rn |     |     |     |     |     |     |     |     |     |

size
32-bit CAS (size == 10 & L == 0 & o0 == 0)
CAS <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASA (size == 10 & L == 1 & o0 == 0)
CASA <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASAL (size == 10 & L == 1 & o0 == 1)
CASAL <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASL (size == 10 & L == 0 & o0 == 1)
CASL <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit CAS (size == 11 & L == 0 & o0 == 0)
CAS <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASA (size == 11 & L == 1 & o0 == 0)
CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASAL (size == 11 & L == 1 & o0 == 1)
CASAL <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASL (size == 11 & L == 0 & o0 == 1)
CASL <Xs>, <Xt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, regsize);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*
For information about memory accesses see *Load/Store addressing modes.*

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

### No offset

(Feat_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 1</td>
</tr>
</tbody>
</table>

**CASAB (L == 1 & o0 == 0)**

CASAB <Ws>, <Wt>, [<Xn|SP>{,#0}]

**CASALB (L == 1 & o0 == 1)**

CASALB <Ws>, <Wt>, [<Xn|SP>{,#0}]

**CASB (L == 0 & o0 == 0)**

CASB <Ws>, <Wt>, [<Xn|SP>{,#0}]

**CASLB (L == 0 & o0 == 1)**

CASLB <Ws>, <Wt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = \texttt{X}[s];
newvalue = \texttt{X}[t];

if n == 31 then
    CheckSPAlignment();
    address = \texttt{SP}[];
else
    address = \texttt{X}[n];

data = \texttt{MemAtomicCompareAndSwap}(address, comparevalue, newvalue, ldacctype, stacctype);
\texttt{X}[s] = \texttt{ZeroExtend}(data, 32);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset
(FEAT_LSE)

```
0 1 0 0 1 0 0 1 1 1 1 1 1
L o0 Rs Rn Rt
```

CASAH (L == 1 && o0 == 0)

CASAH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASALH (L == 1 && o0 == 1)

CASALH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASH (L == 0 && o0 == 0)

CASH <Ws>, <Wt>, [<Xn|SP>{,#0}]

CASLH (L == 0 && o0 == 1)

CASLH <Ws>, <Wt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

```
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != #31;
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);
X[s] = ZeroExtend(data, 32);
CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is \(<W_s>\) and \(<W(s+1)>\), or \(<X_s>\) and \(<X(s+1)>\), are restored to the values held in the registers before the instruction was executed.

No offset

(POSE_LSE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
32-bit CASP (sz == 0 & L == 0 & o0 == 0)
CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPA (sz == 0 & L == 1 & o0 == 0)
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPAL (sz == 0 & L == 1 & o0 == 1)
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPL (sz == 0 & L == 0 & o0 == 1)
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

64-bit CASP (sz == 1 & L == 0 & o0 == 0)
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPA (sz == 1 & L == 1 & o0 == 0)
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPAL (sz == 1 & L == 1 & o0 == 1)
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPL (sz == 1 & L == 0 & o0 == 1)
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
integer datasize = 32 << UInt(sz);
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.

<W(s+1)> Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Wt> Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.

<W(t+1)> Is the 32-bit name of the second general-purpose register to be conditionally stored.

<Xs> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.

<X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian(ldacctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(stacctype) then t1:t2 else t2:t1;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

if BigEndian(ldacctype) then
    X[s] = data<2*datasize-1:datasize>;
    X[s+1] = data<datasize-1:0>;
else
    X[s] = data<datasize-1:0>;
    X[s+1] = data<2*datasize-1:datasize>;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a
PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This
instruction does not affect the condition flags.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 1 1 0 1 0 1</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit \( sf = 0 \)

```markdown
CBNZ \(<Wt>, <label>\)
```

64-bit \( sf = 1 \)

```markdown
CBNZ \(<Xt>, <label>\)
```

```markdown
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

- \(<Wt>\) Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- \(<Xt>\) Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- \(<label>\) Is the program label to be conditionally branched to. Its offset from the address of this instruction, in
  the range +/-1MB, is encoded as "imm19" times 4.

Operation

```markdown
bits(datasize) operand1 = X[t];
if IsZero(operand1) == FALSE then
  BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.16decrcl, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 1 1 0 1 0</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
</table>

op

32-bit (sf == 0)

CBZ <Wt>, <label>

64-bit (sf == 1)

CBZ <Xt>, <label>

integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
b bits(64) offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(datasize) operand1 = X[t];
if IsZero(operand1) == TRUE then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CCMN (immediate)

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

32-bit (sf == 0)

CCMN <Wn>, #<imm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMN <Xn>, #<imm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<imm> Is a five bit unsigned (positive) immediate encoded in the “imm5” field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n];
    (, flags) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>0</th>
<th>nzcv</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CCMN <Wn>, <Wm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMN <Xn>, <Xm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = X[m];
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

32-bit (sf == 0)

CCMP <Wn>, #<imm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMP <Xn>, #<imm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<imm> Is a five bit unsigned (positive) immediate encoded in the “imm5” field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.
<cond> Is one of the standard conditions, encoded in the “cond” field in the standard way.

Operation

if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2;
    operand2 = NOT(imm);
    (., flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>0</th>
<th>nzcv</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CCMP \(<Wn>, <Wm>, #<nzcv>, <cond>\)

64-bit (sf == 1)

CCMP \(<Xn>, <Xm>, #<nzcv>, <cond>\)

```plaintext
ingether n = UInt(Rn);
ingether m = UInt(Rm);
ingether datasize = if sf == '1' then 64 else 32;
ingether flags = nzcv;
```

Assembler Symbols

- \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<nzcv>\) Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- \(<cond>\) Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```plaintext
if ConditionHolds(cond) then
  bits(datasize) operand1 = X[n];
  bits(datasize) operand2 = X[m];
  operand2 = NOT(operand2);
  (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System

(FEAT_FlagM)

```
0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 |(0)(0)(0)(0)| 0 0 0 | 1 1 1 1 1
```

**CFINV**

if !HaveFlagManipulateExt() then UNDEFINED;

**Operation**

PSTATE.C = NOT(PSTATE.C);

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see **CFP RCTX, Control Flow Prediction Restriction by Context**.

This is an alias of **SYS**. This means:

- The encodings in this description are named to match the encodings of **SYS**.
- The description of **SYS** gives the operational pseudocode for this instruction.

**System**

(Feat_SpecRes)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CFP RCTX, <Xt>**

is equivalent to

**SYS #3, C7, C3, #4, <Xt>**

and is always the preferred disassembly.

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

The description of **SYS** gives the operational pseudocode for this instruction.
CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CINC. This means:

- The encodings in this description are named to match the encodings of CINC.
- The description of CINC gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CINC <Wd>, <Wn>, <cond>

is equivalent to

CINC <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CINC <Xd>, <Xn>, <cond>

is equivalent to

CINC <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CINV

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINV. This means:

• The encodings in this description are named to match the encodings of CSINV.
• The description of CSINV gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CINV <Wd>, <Wn>, <cond>

is equivalent to

CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CINV <Xd>, <Xn>, <cond>

is equivalent to

CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
CLREX

Clear Exclusive clears the local monitor of the executing PE.

\[
\begin{array}{cccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & CRm & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

CLREX {\#<imm>}

// CRm field is ignored

Assembler Symbols

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

\texttt{ClearExclusiveLocal(ProcessorID());}

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

**32-bit (sf == 0)**

CLS <Wd>, <Wn>

**64-bit (sf == 1)**

CLS <Xd>, <Xn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

```plaintext
integer result;
bits(datasize) operand1 = X[n];
result = CountLeadingSignBits(operand1);
X[d] = result<datasize-1:0>;
```

**Operational Information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CLZ

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.

32-bit (sf == 0)

CLZ <Wd>, <Wn>

64-bit (sf == 1)

CLZ <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

integer result;
bits(datasize) operand1 = X[n];
result = CountLeadingZeroBits(operand1);
X[d] = result<datasize-1:0>;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**CMN (extended register)**

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the \(<Rm>\) register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of **ADDS (extended register)**. This means:

- The encodings in this description are named to match the encodings of **ADDS (extended register)**.
- The description of **ADDS (extended register)** gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

\[ \text{CMN \,<Wn|WSP>, \,<Wm>{, \,<extend> \{#<amount>\}}} \]

is equivalent to

\[ \text{ADDS \,WZR, \,<Wn|WSP>, \,<Wm>{, \,<extend> \{#<amount>\}}} \]

and is always the preferred disassembly.

### 64-bit (sf == 1)

\[ \text{CMN \,<Xn|SP>, \,<R><m>{, \,<extend> \{#<amount>\}}} \]

is equivalent to

\[ \text{ADDS \,XZR, \,<Xn|SP>, \,<R><m>{, \,<extend> \{#<amount>\}}} \]

and is always the preferred disassembly.

**Assembler Symbols**

- **<Wn|WSP>** Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xn|SP>** Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- **<R>** Is a width specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- **<m>** Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- **<extend>** For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":
If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '1111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

ADDS WZR, <Wn|WSP>, #<imm> {, <shift>}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn|SP>, #<imm>{, <shift>}

is equivalent to

ADDS XZR, <Xn|SP>, #<imm> {, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP>  Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xn|SP>    Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>      Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>    Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMN (shifted register)

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

| sf | 0 | 1 | 0 | 1 | 0 | 1 | 1 | shift | 0 | Rm | imm6 | Rn | 1 | 1 | 1 | 1 | Rd |
|----|---|---|---|---|---|---|---|-------|---|----|------|----|---|---|---|---|
| op | S |

32-bit (sf == 0)

CMN <Wn>, <Wm>{, <shift> #<amount>}

is equivalent to

ADDS WZR, <Wn>, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn>, <Xm>{, <shift> #<amount>}

is equivalent to

ADDS XZR, <Xn>, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMP (extended register)

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (extended register). This means:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMP <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

is equivalent to

SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

and is always the preferred disassembly.

64-bit (sf == 1)

CMP <Xn|SP>, <R><m>{, <extend> {#<amount>}}

is equivalent to

SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option".
If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If “Rn” is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

 CMP <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

 SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}

and is always the preferred disassembly.

64-bit (sf == 1)

 CMP <Xn|SP>, #<imm>{, <shift>}

is equivalent to

 SUBS XZR, <Xn|SP>, #<imm> {, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**CMP (shifted register)**

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>op</th>
<th>Rd</th>
<th>imm6</th>
<th>Rn</th>
<th>shift</th>
<th>Rm</th>
<th>sf</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>0111</td>
<td>1</td>
<td>1000</td>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

CMP <Wn>, <Wm>{, <shift> #<amount>}

is equivalent to

SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

**64-bit (sf == 1)**

CMP <Xn>, <Xm>{, <shift> #<amount>}

is equivalent to

SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

**Assembler Symbols**

- <Wn>: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm>: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xn>: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm>: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift>: Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- <amount>: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of SUBPS. This means:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode for this instruction.

### Integer
(Armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 0 1 1 1 0 1 0 1 1 0</th>
<th>0 0 0 0 0</th>
<th>1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xn</td>
<td>Xm</td>
<td>Xn</td>
<td>Xd</td>
</tr>
</tbody>
</table>

CMPP `<Xn|SP>`, `<Xm|SP>`

is equivalent to

SUBPS XZR, `<Xn|SP>`, `<Xm|SP>`

and is always the preferred disassembly.

### Assembler Symbols

- `<Xn|SP>`: Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Xn” field.
- `<Xm|SP>`: Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Xm” field.

### Operation

The description of SUBPS gives the operational pseudocode for this instruction.
CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CNEG <Wd>, <Wn>, <cond> is equivalent to

CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CNEG <Xd>, <Xn>, <cond> is equivalent to

CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Cache prefetch predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot influence speculative execution occurring after the instruction is complete and synchronized.

For more information, see **CPP RCTX, Cache Prefetch Prediction Restriction by Context**.

This is an alias of **SYS**. This means:

- The encodings in this description are named to match the encodings of **SYS**.
- The description of **SYS** gives the operational pseudocode for this instruction.

### System

(\texttt{FEAT\_SPECRES})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  |

**CPP RCTX, \langle X_t \rangle**

is equivalent to

**SYS #3, C7, C3, #7, \langle X_t \rangle**

and is always the preferred disassembly.

### Assembler Symbols

\langle X_t \rangle \quad \text{Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.}

### Operation

The description of **SYS** gives the operational pseudocode for this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CPYFP, CPYFM, CPYFE

Memory Copy Forward-only. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE. CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFE performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz  0 1 1 0 0 1 op1 0 Rs  0 0 0 0 0 1 Rn</td>
</tr>
<tr>
<td>op2</td>
</tr>
</tbody>
</table>

Epilogue (op1 == 10)

CPYFE [Xd]!, [Xs]!, <Xn>!

Main (op1 == 01)

CPYFM [Xd]!, [Xs]!, <Xn>!

Prologue (op1 == 00)

CPYFP [Xd]!, [Xs]!, <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

  if supports option a then
    PSTATE.C = '0';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize = Zeros();

    if SInt(cpysize) > 0 then
      assert SInt(stagecpysize) <= SInt(cpysize);
    else
      assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if PSTATE.C == '0' then
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSSStage_Main then
      stagecpysize = cpysize - postsize;

      // Check if the parameters to this instruction are valid.
      if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

      else
        stagecpysize = postsize;

      // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= -1 * SInt(stagecpysize);
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    cpysize = cpysize + B;
    stagecpysize = stagecpysize + B;

    if stage != MOPSSStage_Prologue then
      X[n] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage_Prologue then
          X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
      end
    end
  end
end
Memory Copy Forward-only, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPN, option A (which results in encoding PSTATE.C = 0):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + saturated Xn.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPN, option B (which results in encoding PSTATE.C = 1):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
• Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
• Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ◦ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  ◦ the value of Xs is written back with the lowest address that has not been copied from.
  ◦ the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ◦ the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value of Xd is written back with the lowest address that has not been copied to.

---

**Integer**

( **FEAT_MOPS** )

| sz | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op2 | 0 | 1 | 1 | 0 | 0 | 1 | op1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

**Epilogue (op1 == 10)**

CPYFEN [<Xd>], [<Xs>], <Xn>!

**Main (op1 == 01)**

CPYFMN [<Xd>], [<Xs>], <Xn>!

**Prologue (op1 == 00)**

CPYFPN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

**MOPSSStage** stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

**Assembler Symbols**

**<Xd>**

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

**<Xs>**

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

**<Xn>**

For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
        if SInt(cpysize) > 0 then
            assert SInt(stagecpysize) <= SInt(cpysize);
        else
            assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                boolean zero_size_exceptions = MemCpyZeroSizeCheck();
        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

        if stage == MOPSSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
               // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || \texttt{MemCpyParametersIllformedE}(toaddress, fromaddress, cpysize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  \texttt{MismatchedMemCpyException}(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
  while \texttt{SInt}(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = \texttt{CPYSizeChoice}(toaddress, fromaddress, cpysize);
    assert B <= -1 * \texttt{SInt}(stagecpysize);

    readdata<8\cdot B-1:0> = \texttt{Mem}[fromaddress+cpysize, B, racctype];
    \texttt{Mem}[toaddress+cpysize, B, wacctype] = readdata<8\cdot B-1:0>;
    cpysize = cpysize + B;
    stagecpysize = stagecpysize + B;

    if stage != \texttt{MOPSSStage_Prologue} then
      \texttt{X}[n] = cpysize;
    else
      while \texttt{UInt}(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = \texttt{CPYSizeChoice}(toaddress, fromaddress, cpysize);
        assert B <= \texttt{UInt}(stagecpysize);

        readdata<8\cdot B-1:0> = \texttt{Mem}[fromaddress, B, racctype];
        \texttt{Mem}[toaddress, B, wacctype] = readdata<8\cdot B-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != \texttt{MOPSSStage_Prologue} then
          \texttt{X}[n] = cpysize;
          \texttt{X}[d] = toaddress;
          \texttt{X}[s] = fromaddress;
        if stage == \texttt{MOPSSStage_Prologue} then
          \texttt{X}[n] = cpysize;
          \texttt{X}[d] = toaddress;
          \texttt{X}[s] = fromaddress;

CPYFPRN, CPYFMRN, CPYFERN

Memory Copy Forward-only, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRN, option A (which results in encoding PSTATE.C = 0):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + saturated Xn.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.(N,Z,V) are set to {0,0,0}.

After execution of CPYFPRN, option B (which results in encoding PSTATE.C = 1):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
• Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
• Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.(N,Z,V) are set to {0,0,0}.

For CPYFMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ◦ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  ◦ the value of Xs is written back with the lowest address that has not been copied from.
  ◦ the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ◦ the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value of Xd is written back with the lowest address that has not been copied to.

Integer
(Feat_Mops)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 0  | 0  | 1  |    | op1|    | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rs | 1  | 0  | 0  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Epilogue (op1 == 10)

CPYFERN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYFMRN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYFPRN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMops() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSstage stage;
case op1 of
  when '00' stage = MOPSSstage_Prologue;
  when '01' stage = MOPSSstage_Main;
  when '10' stage = MOPSSstage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFF<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

                readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
                Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
                fromaddress = fromaddress + B;
                toaddress = toaddress + B;

                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;

                if stage != MOPSSStage_Prologue then
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;
                else
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
Memory Copy Forward-only, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and performs an implementation-defined amount of the memory copy. CPYFMRT performs an implementation-defined amount of the memory copy. CPYFERT performs the last part of the memory copy.

**Note**

The inclusion of implementation-defined amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation-defined.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRT, option A (which results in encoding PSTATE.C = 0):

- If \( Xn<63> == 1 \), the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- \( Xs \) holds the original \( Xs + \) saturated \( Xn \).
- \(Xd\) holds the original \( Xd + \) saturated \( Xn \).
- \(Xn\) holds \(-1\) saturated \( Xn + \) an implementation-defined number of bytes copied.
- PSTATE.(N,Z,V) are set to \( \{0,0,0\} \).

After execution of CPYFPRT, option B (which results in encoding PSTATE.C = 1):

- If \( Xn<63> == 1 \), the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- \( Xs \) holds the original \( Xs + \) an implementation-defined number of bytes copied.
- \(Xd\) holds the original \( Xd + \) an implementation-defined number of bytes copied.
- \(Xn\) holds the saturated \( Xn - \) an implementation-defined number of bytes copied.
- PSTATE.(N,Z,V) are set to \( \{0,0,0\} \).

For CPYFMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- \( Xn \) is treated as a signed 64-bit number and holds \(-1\) the number of bytes remaining to be copied in the memory copy in total.
- \( Xs \) holds the lowest address that the copy is copied from -\( Xn \).
- \(Xd\) holds the lowest address that the copy is made to -\( Xn \).
- At the end of the instruction, the value of \( Xn \) is written back with \(-1\) the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- \( Xn \) holds the number of bytes remaining to be copied in the memory copy in total.
- \( Xs \) holds the lowest address that the copy is copied from.
- \(Xd\) holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of \( Xn \) is written back with \( Xn \).
  - the value of \( Xs \) is written back with the lowest address that has not been copied from.
  - the value of \( Xd \) is written back with the lowest address that has not been copied to.

For CPYFERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- \( Xn \) is treated as a signed 64-bit number and holds \(-1\) the number of bytes remaining to be copied in the memory copy in total.
- \( Xs \) holds the lowest address that the copy is copied from -\( Xn \).
- \(Xd\) holds the lowest address that the copy is made to -\( Xn \).
- At the end of the instruction, the value of \( Xn \) is written back with 0.

For CPYFERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- \( Xn \) holds the number of bytes remaining to be copied in the memory copy in total.
- \( Xs \) holds the lowest address that the copy is copied from.
- \(Xd\) holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of \( Xn \) is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value ofXd is written back with the lowest address that has not been copied to.

Integer
(Feat_Mops)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Epilogue (op1 == 10)

CPYFERT [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYFMRT [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYFPRT [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
  For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

// IMP DEF selection of the amount covered by pre-processing.
stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;
                // Check if the parameters to this instruction are valid for the epilogue.
        else
            stagecpysize = postsize;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

        if stage == MOPSStage_Epilogue then
            stagecpysize = postsize;
            // Check if the parameters to this instruction are valid for the epilogue.
        else
            stagecpysize = postsize;
            // Check if the parameters to this instruction are valid for the epilogue.

    else
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while $\text{SInt}$(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * $\text{SInt}$(stagecpysize);

        readdata<8*-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<8*-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
        if stage != MOPSSStage_Prologue then
            \( \mathbf{X} \)[n] = cpysize;
        end
    end
else
    while $\text{UInt}$(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= $\text{UInt}$(stagecpysize);

        readdata<8*-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<8*-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        if stage != MOPSSStage_Prologue then
            \( \mathbf{X} \)[n] = cpysize;
            \( \mathbf{X} \)[d] = toaddress;
            \( \mathbf{X} \)[s] = fromaddress;
        end
    end
if stage == MOPSSStage_Prologue then
    \( \mathbf{X} \)[n] = cpysize;
    \( \mathbf{X} \)[d] = toaddress;
    \( \mathbf{X} \)[s] = fromaddress;

Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value ofXd is written back with the lowest address that has not been copied to.

### Integer

(Feat_Mops)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz  | 0  | 1  | 1  | 0  | 0  | 1  | op1| 0  | Rs | 1  | 1  | 1  | 0  | 0  | 1  | Rn | Rd |

### Epilogue (op1 == 10)

CPYFERTN [<Xd>], [<Xs>], <Xn>!

### Main (op1 == 01)

CPYFMRTN [<Xd>], [<Xs>], <Xn>!

### Prologue (op1 == 00)

CPYFPRTN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMops() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

### Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

        if SInt(cpysize) > 0 then
            assert SInt(stagecpysize) <= SInt(cpysize);
        else
            assert SInt(stagecpysize) >= SInt(cpysize);

    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

            else
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

        if stage == MOPSSStage_Main then
            stagecpysize = cpysize - postsize;

            // Check if the parameters to this instruction are valid.
            if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= -1 * SInt(stagecpysize);
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    cpysize = cpysize + B;
    stagecpysize = stagecpysize + B;
    if stage != MOPSSStage_Prologue then
      X[n] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        if stage != MOPSSStage_Prologue then
          X[n] = cpysize;
          X[d] = toaddress;
          X[s] = fromaddress;
        else
          X[n] = cpysize;
          X[d] = toaddress;
          X[s] = fromaddress;
          if stage == MOPSSStage_Prologue then
            X[n] = cpysize;
            X[d] = toaddress;
            X[s] = fromaddress;
Memory Copy Forward-only, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN. CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTRN performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTRN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFF
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTRN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFF
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.

For CPYFERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTRN option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

### Integer

**FEAT_MOPS**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 0  | 0  | 1  | op1| 0  | Rs | 1  | 0  | 1  | 0  | 0  | 1  | Rn | Rd | op2|

#### Epilogue (op1 == 10)

`CPYFERTRN [<Xd>], [<Xs>], <Xn>`!

#### Main (op1 == 01)

`CPYFMTRRN [<Xd>], [<Xs>], <Xn>`!

#### Prologue (op1 == 00)

`CPYFPTRRN [<Xd>], [<Xs>], <Xn>`!

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

### Assembler Symbols

- `<Xd>` For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
  
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

- `<Xs>` For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
  
  For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

- `<Xn>` For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
  
  For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

  For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFF<63:0>;
  if supports_option_a then
    PSTATE.C = '0';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';
  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();
  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if PSTATE.C == '1' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if PSTATE.C == '0' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
  assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
  if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;
    // Check if the parameters to this instruction are valid.
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = FALSE;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      stagecpysize = postsize;
      // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || \texttt{MemCpyParametersIllformedE}(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    \texttt{MismatchedMemCpyException}(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while \texttt{SInt}(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = \texttt{CPYSizeChoice}(toaddress, fromaddress, cpysize);
        assert B <= -1 * \texttt{SInt}(stagecpysize);
        readdata<B*8-1:0> = \texttt{Mem}[fromaddress+cpysize, B, racctype];
        \texttt{Mem}[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
        if stage != \texttt{MOPSSStage_Prologue} then
            \texttt{X}[n] = cpysize;
        end if
    end while
else
    while \texttt{UInt}(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = \texttt{CPYSizeChoice}(toaddress, fromaddress, cpysize);
        assert B <= \texttt{UInt}(stagecpysize);
        readdata<B*8-1:0> = \texttt{Mem}[fromaddress, B, racctype];
        \texttt{Mem}[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        if stage != \texttt{MOPSSStage_Prologue} then
            \texttt{X}[n] = cpysize;
            \texttt{X}[d] = toaddress;
            \texttt{X}[s] = fromaddress;
        end if
    end while
end if
\texttt{X}[n] = cpysize;
\texttt{X}[d] = toaddress;
\texttt{X}[s] = fromaddress;
**CPYFPRTWN, CPYFMRTWN, CPYFERTWN**

Memory Copy Forward-only, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTWN performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTWN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
-Xd holds the original Xd + saturated Xn.
-Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTWN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
-Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
-Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
-Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
-Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
-Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
-Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

### Integer (FEAT_MOPS)

<table>
<thead>
<tr>
<th></th>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

### Epilogue (op1 == 10)

CPYFERTWN [<Xd>], [<Xs>], <Xn>!

### Main (op1 == 01)

CPYFMRTWN [<Xd>], [<Xs>], <Xn>!

### Prologue (op1 == 00)

CPYFPRTWN [<Xd>], [<Xs>], <Xn>!

If !HaveFeatMOPS() then UNDEFINED;
If sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;

Case op1 of
    When '00' stage = MOPSSStage_Prologue;
    When '01' stage = MOPSSStage_Main;
    When '10' stage = MOPSSStage_Epilogue;
    Otherwise SEE "Memory Copy and Memory Set";

If d == s || s == n || d == n then UNDEFINED;
If d == 31 || s == 31 || n == 31 then UNDEFINED;

### Assembler Symbols

**<Xd>**
- For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

**<Xs>**
- For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

**<Xn>**
- For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
integer \( N = \text{MaxBlockSizeCopiedBytes}() \);

bits(64) toaddress = \( X[d] \);
bits(64) fromaddress = \( X[s] \);
bits(64) cpysize = \( X[n] \);

bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if \( \text{HaveMTE2Ext}() \) then
    \( \text{SetTagCheckedInstruction}(\text{TRUE}) \);

boolean supports_option_a = \( \text{MemCpyOptionA}() \);
(racctype, wacctype) = \( \text{MemCpyAccessTypes}(\text{options}) \);

if stage == \text{MOPSSStage_Prologue} then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = \( \text{Zeros}(64) - \text{cpysize} \);
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = \( \text{CPYPreSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == \( \text{Zeros}(64) \);

        if \( \text{SInt}(\text{cpysize}) > 0 \) then
            assert \( \text{SInt}(\text{stagecpysize}) \leq \text{SInt}(\text{cpysize}) \);
        else
            assert \( \text{SInt}(\text{stagecpysize}) \geq \text{SInt}(\text{cpysize}) \);

        assert stagecpysize<63> == \( \text{SInt}(\text{stagecpysize}) \);
    else
        boolean zero_size_exceptions = \( \text{MemCpyZeroSizeCheck}() \);

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || \( \text{SInt}(\text{cpysize}) != 0 \) then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = \text{TRUE};
                    boolean from_epilogue = stage == \text{MOPSSStage_Epilogue};
                    \( \text{MismatchedMemCpyException}(\text{supports_option_a, d, s, n, wrong_option, from_epilogue, options}) \);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = \text{TRUE};
                        boolean from_epilogue = stage == \text{MOPSSStage_Epilogue};
                        \( \text{MismatchedMemCpyException}(\text{supports_option_a, d, s, n, wrong_option, from_epilogue, options}) \);

                bits(64) postsize = \( \text{CPYPostSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);
                assert postsize<63> == cpysize<63> || \( \text{SInt}(\text{postsize}) == 0 \);

                if stage == \text{MOPSSStage_Main} then
                    stagecpysize = cpysize - postsize;

                    // Check if the parameters to this instruction are valid.
                    if \( \text{MemCpyParametersIllformed}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \) then
                        boolean wrong_option = \text{FALSE};
                        boolean from_epilogue = \text{FALSE};
                        \( \text{MismatchedMemCpyException}(\text{supports_option_a, d, s, n, wrong_option, from_epilogue, options}) \);
                    else
                        stagecpysize = postsize;

                    // Check if the parameters to this instruction are valid for the epilogue.
            else
                \( \text{MismatchedMemCpyException}(\text{supports_option_a, d, s, n, wrong_option, from_epilogue, options}) \);
        else
            \( \text{MismatchedMemCpyException}(\text{supports_option_a, d, s, n, wrong_option, from_epilogue, options}) \);

        bits(64) postsize = \( \text{CPYPostSizeChoice}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \);
        assert postsize<63> == \( \text{SInt}(\text{postsize}) == 0 \);

        if stage == \text{MOPSSStage_Main} then
            stagecpysize = cpysize - postsize;

            // Check if the parameters to this instruction are valid.
            if \( \text{MemCpyParametersIllformed}(\text{toaddress}, \text{fromaddress}, \text{cpysize}) \) then
                boolean wrong_option = \text{FALSE};
                boolean from_epilogue = \text{FALSE};
                \( \text{MismatchedMemCpyException}(\text{supports_option_a, d, s, n, wrong_option, from_epilogue, options}) \);
            else
                stagecpysize = postsize;

                // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || \texttt{MemCpyParametersIllformedE}(toaddress, fromaddress, cpysize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  \texttt{MismatchedMemCpyException}(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
  while \texttt{SInt}(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = \texttt{CPYSIZEChoice}(toaddress, fromaddress, cpysize);
    assert B <= -1 * \texttt{SInt}(stagecpysize);

    readdata\langle B*8-1:0 \rangle = \texttt{Mem}[fromaddress+cpysize, B, racctype];
    \texttt{Mem}[toaddress+cpysize, B, wacctype] = readdata\langle B*8-1:0 \rangle;
    cpysize = cpysize + B;
    stagecpysize = stagecpysize + B;

    if stage != \texttt{MOPSSStage_Prologue} then
      X[n] = cpysize;
  else
    while \texttt{UInt}(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = \texttt{CPYSIZEChoice}(toaddress, fromaddress, cpysize);
      assert B <= \texttt{UInt}(stagecpysize);

      readdata\langle B*8-1:0 \rangle = \texttt{Mem}[fromaddress, B, racctype];
      \texttt{Mem}[toaddress, B, wacctype] = readdata\langle B*8-1:0 \rangle;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;

      if stage != \texttt{MOPSSStage_Prologue} then
        X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
      else
        X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
Memory Copy Forward-only, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFET performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPT, option A (which results in encoding PSTATE.C = 0):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + saturated Xn.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPT, option B (which results in encoding PSTATE.C = 1):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
• Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
• Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
• PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ◦ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  ◦ the value of Xs is written back with the lowest address that has not been copied from.
  ◦ the value of Xd is written back with the lowest address that has not been copied to.

For CPYFET, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from -Xn.
• Xd holds the lowest address that the copy is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For CPYFET, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be copied in the memory copy in total.
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ◦ the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value of Xd is written back with the lowest address that has not been copied to.

Integer
(Feat_MOPS)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | sz | 0 1 1 0 0 1 | op1 | 0 | Rs | 0 0 1 1 0 1 | Rn | Rd |
|---------------------------------------------|----|-------------|-----|----|---------|----|-----|

Epilogue (op1 == 10)

CPYFET [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYFMT [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYFPT [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;
  if supports_option_a then
    PSTATE.C = '0';
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';

stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize = Zeros();

if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else

boolean zero_size_exceptions = MemCpyZeroSizeCheck();

if zero_size_exceptions || SInt(cpysize) != 0 then
  if supports_option_a then
    if PSTATE.C == '1' then
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSSStage_Epilogue;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      if PSTATE.C == '0' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSSStage_Main then
  stagecpysize = cpysize - postsize;

  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

if stage == MOPSSStage_Main then
  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

// Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformed(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

                readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
                Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
                fromaddress = fromaddress + B;
                toaddress = toaddress + B;

                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;

                if stage != MOPSSStage_Prologue then
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;
                else
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;

if stage == MOPSSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
Memory Copy Forward-only, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and performs an implementation defined amount of the memory copy. CPYFMTN performs an implementation defined amount of the memory copy. CPYFETN performs the last part of the memory copy.

**Note**

The inclusion of implementation defined amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation defined.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an implementation defined number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

After execution of CPYFPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an implementation defined number of bytes copied.
- Xd holds the original Xd + an implementation defined number of bytes copied.
- Xn holds the saturated Xn - an implementation defined number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

For CPYFMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.

the value ofXd is written back with the lowest address that has not been copied to.

---

**Integer (FEAT_MOPS)**

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Epilogue (op1 == 10)**

CPYFETN [<Xd>], [<Xs>], <Xn>!

**Main (op1 == 01)**

CPYFMTN [<Xd>], [<Xs>], <Xn>!

**Prologue (op1 == 00)**

CPYFPTN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

---

**Assembler Symbols**

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

  if supports_option_a then
    PSTATE.C = '0';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

    if SInt(cpysize) > 0 then
      assert SInt(stagecpysize) <= SInt(cpysize);
    else
      assert SInt(stagecpysize) >= SInt(cpysize);
  
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if PSTATE.C == '1' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = FALSE;
          boolean from_epilogue = FALSE;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          boolean wrong_option = FALSE;
          boolean from_epilogue = FALSE;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to this instruction are valid for the epilogue.
if (cpsize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpsize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpsize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpsize);
    assert B <= -1 * SInt(stagecpsize);
    readdata<B*8-1:0> = Mem[fromaddress+cpsize, B, racctype];
    Mem[toaddress+cpsize, B, wacctype] = readdata<B*8-1:0>;
    cpsize = cpsize + B;
    stagecpsize = stagecpsize + B;
    if stage != MOPSSStage_Prologue then
      X[n] = cpsize;
    else
      while UInt(stagecpsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpsize);
        assert B <= UInt(stagecpsize);
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpsize = cpsize - B;
        stagecpsize = stagecpsize - B;
        if stage != MOPSSStage_Prologue then
          X[n] = cpsize;
          X[d] = toaddress;
          X[s] = fromaddress;
        else
          X[n] = cpsize;
          X[d] = toaddress;
          X[s] = fromaddress;
CPYFPTRN, CPYFMTRN, CPYFETRN

Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTRN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

After execution of CPYFPTRN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

For CPYFMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

### Integer

**(FEAT_MOPS)**

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
<th>op2</th>
</tr>
</thead>
</table>

**Epilogue (op1 == 10)**

CPYFETRN [<Xd>], [<Xs>], <Xn>!

**Main (op1 == 01)**

CPYFMTRN [<Xd>], [<Xs>], <Xn>!

**Prologue (op1 == 00)**

CPYFPTRN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

**MOPSSStage** stage;

**case op1 of**

- when '00' stage = MOPSSStage_Prologue;
- when '01' stage = MOPSSStage_Main;
- when '10' stage = MOPSSStage_Epilogue;
- otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

### Assembler Symbols

**<Xd>**

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

**<Xs>**

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

**<Xn>**

For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;

if (haveMTE2Ext) then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

if supports_option_a then
  PSTATE.C = '0';
  // Copy in the forward direction offsets the arguments.
  toaddress = toaddress + cpysize;
  fromaddress = fromaddress + cpysize;
  cpysize = Zeros(64) - cpysize;
else
  PSTATE.C = '1';
  PSTATE.N = '0';
  PSTATE.V = '0';
  PSTATE.Z = '0';

// IMP DEF selection of the amount covered by pre-processing.
stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

if (SInt(cpysize) > 0) then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();
  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || (SInt(cpysize) != 0) then
    if supports_option_a then
      if PSTATE.C == '1' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if PSTATE.C == '0' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
      assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
endif

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
        if stage != MOPSStage_Prologue then
            X[n] = cpysize;
        endif
    endwhile
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        if stage != MOPSStage_Prologue then
            X[n] = cpysize;
            X[d] = toaddress;
            X[s] = fromaddress;
        endif
    endwhile
endif

if stage == MOPSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
endif
CPYFPTWN, CPYFMTWN, CPYFETWN

Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTWN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTWN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

### Integer (FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Epilogue (op1 == 10)

CPYFETWN [<Xd>], [<Xs>], <Xn>!

#### Main (op1 == 01)

CPYFMFTWN [<Xd>], [<Xs>], <Xn>!

#### Prologue (op1 == 00)

CPYFPTWN [<Xd>], [<Xs>], <Xn>!

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
   when '00' stage = MOPSSStage_Prologue;
   when '01' stage = MOPSSStage_Main;
   when '10' stage = MOPSSStage_Epilogue;
   otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

### Assembler Symbols

- **<Xd>**: For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

- **<Xs>**: For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

- **<Xn>**: For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0xFFFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);

else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
        end if
    end while

else

    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
            X[d] = toaddress;
            X[s] = fromaddress;
        end if
    end while

if stage == MOPSSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
end if
Memory Copy Forward-only, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN, then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and performs an implementation defined amount of the memory copy. CPYFMWN performs an implementation defined amount of the memory copy. CPYFEWN performs the last part of the memory copy.

Note

The inclusion of implementation defined amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation defined.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an implementation defined number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an implementation defined number of bytes copied.
- Xd holds the original Xd + an implementation defined number of bytes copied.
- Xn holds the saturated Xn - an implementation defined number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
the value of $X_s$ is written back with the lowest address that has not been copied from.
the value of $X_d$ is written back with the lowest address that has not been copied to.

Integer
(Feat_Mops)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 0  | 0  | 1  | op1| 0  | Rs | 0  | 1  | 0  | 0  | 1  | Rn | Rd |
| op2|

Epilogue (op1 == 10)

CPYFEWN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYFMWN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYFPWN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0xFFFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

        if SInt(cpysize) > 0 then
            assert SInt(stagecpysize) <= SInt(cpysize);
        else
            assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;

            // Check if the parameters to this instruction are valid for the epilogue.
            CPYFPWN, CPYFMWN,
            CPYFEWN
if (cpysize != postsize || 
   MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then 
   boolean wrong_option = FALSE; 
   boolean from_epilogue = TRUE; 
   MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options); 

if supports_option_a then 
   while SInt(stagecpysize) != 0 do 
      // IMP DEF selection of the block size that is worked on. While many 
      // implementations might make this constant, that is not assumed. 
      B = CPYSizeChoice(toaddress, fromaddress, cpysize); 
      assert B <= -1 * SInt(stagecpysize); 
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype]; 
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>; 
      cpysize = cpysize + B; 
      stagecpysize = stagecpysize + B; 
      if stage != MOPSStage_Prologue then 
         X[n] = cpysize; 
      else 
         while UInt(stagecpysize) > 0 do 
            // IMP DEF selection of the block size that is worked on. While many 
            // implementations might make this constant, that is not assumed. 
            B = CPYSizeChoice(toaddress, fromaddress, cpysize); 
            assert B <= UInt(stagecpysize); 
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype]; 
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>; 
            fromaddress = fromaddress + B; 
            toaddress = toaddress + B; 
            cpysize = cpysize - B; 
            stagecpysize = stagecpysize - B; 
            if stage != MOPSStage_Prologue then 
               X[n] = cpysize; 
               X[d] = toaddress; 
               X[s] = fromaddress; 
            if stage == MOPSStage_Prologue then 
               X[n] = cpysize; 
               X[d] = toaddress; 
               X[s] = fromaddress;
Memory Copy Forward-only, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value ofXd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz  | 0  | 1  | 1  | 0  | 0  | 1  | op1| 0  | Rs | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| op2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Epilogue (op1 == 10)

`CPYFEWT [Xd]!, [Xs]!, Xn!`

Main (op1 == 01)

`CPYFMWT [Xd]!, [Xs]!, Xn!`

Prologue (op1 == 00)

`CPYFPWT [Xd]!, [Xs]!, Xn!`

if `!HaveFeatMOPS()` then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);

`MOPSSStage` stage;

case op1 of
  when '00' stage = `MOPSSStage_Prologue`;
  when '01' stage = `MOPSSStage_Main`;
  when '10' stage = `MOPSSStage_Epilogue`;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation

CPYFPWT, CPYFMWT, CPYFEWT
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpsize = X[n];
bits(64) stagecpsize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpsize<63> == '1' then cpsize = 0x7FFFFFFFFFFFFFF<63:0>;

  if supports_option_a then
    PSTATE.C = '0';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpsize;
    fromaddress = fromaddress + cpsize;
    cpsize = Zeros(64) - cpsize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpsize = CPYPreSizeChoice(toaddress, fromaddress, cpsize);
    assert stagecpsize<63> == cpsize<63> || stagecpsize == Zeros();

    if SInt(cpsize) > 0 then
      assert SInt(stagecpsize) <= SInt(cpsize);
    else
      assert SInt(stagecpsize) >= SInt(cpsize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpsize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpsize);
    assert postsize<63> == cpsize<63> || SInt(postsize) == 0;

    if stage == MOPSSStage_Main then
      stagecpsize = cpsize - postsize;

      // Check if the parameters to this instruction are valid.
      if MemCpyParametersIllformedM(toaddress, fromaddress, cpsize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        stagecpsize = postsize;

    // Check if the parameters to this instruction are valid for the epilogue.
}

CPYFPWT, CPYFMWT, CPYFEWT
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= -1 * SInt(stagecpysize);
    readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
    Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    cpysize = cpysize + B;
    stagecpysize = stagecpysize + B;
    if stage != MOPSSStage_Prologue then
      X[n] = cpysize;
  end while
else
  while UInt(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecpysize);
    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
    fromaddress = fromaddress + B;
    toaddress = toaddress + B;
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
    if stage != MOPSSStage_Prologue then
      X[n] = cpysize;
      X[d] = toaddress;
      X[s] = fromaddress;
    end if
  end while
if stage == MOPSSStage_Prologue then
  X[n] = cpysize;
  X[d] = toaddress;
  X[s] = fromaddress;

Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and performs an implementation defined amount of the memory copy. CPYFMWTN performs an implementation defined amount of the memory copy. CPYFEWTN performs the last part of the memory copy.

Note

The inclusion of implementation defined amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation defined.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an implementation defined number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

After execution of CPYFPWTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an implementation defined number of bytes copied.
- Xd holds the original Xd + an implementation defined number of bytes copied.
- Xn holds the saturated Xn - an implementation defined number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

For CPYFMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value of Xd is written back with the lowest address that has not been copied to.

Integer
(\texttt{FEAT\_MOPS})

<table>
<thead>
<tr>
<th>\texttt{sz}</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>\texttt{op1}</th>
<th>0</th>
<th>\texttt{Rs}</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>\texttt{Rn}</th>
<th>\texttt{Rd}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{op2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Epilogue (\texttt{op1} == 10)

\texttt{CPYFEWTN} \([\langle Xd \rangle], [\langle Xs \rangle], [\langle Xn \rangle])

Main (\texttt{op1} == 01)

\texttt{CPYFMWTN} \([\langle Xd \rangle], [\langle Xs \rangle], [\langle Xn \rangle])

Prologue (\texttt{op1} == 00)

\texttt{CPYFPWTN} \([\langle Xd \rangle], [\langle Xs \rangle], [\langle Xn \rangle])

\text{if} \ \texttt{!HaveFeatMOPS()} \text{then UNDEFINED;}
\text{if} \ \texttt{sz} \ != \ '00' \ \text{then UNDEFINED;}

\text{integer} \ d = \texttt{UInt}(\texttt{Rd});
\text{integer} \ s = \texttt{UInt}(\texttt{Rs});
\text{integer} \ n = \texttt{UInt}(\texttt{Rn});
\text{bits(4)} \ \text{options} = \texttt{op2};

\text{MOPSSStage} \ \text{stage;}
\text{case} \ \texttt{op1} \ \text{of}
\hspace{1em} \text{when} \ '00' \ \text{stage} = \text{MOPSSStage\_Prologue;}
\hspace{1em} \text{when} \ '01' \ \text{stage} = \text{MOPSSStage\_Main;}
\hspace{1em} \text{when} \ '10' \ \text{stage} = \text{MOPSSStage\_Epilogue;}
\hspace{1em} \text{others} \ \text{wise} \ \text{SEE "Memory Copy and Memory Set";}

\text{if} \ d == s || s == n || d == n \ \text{then UNDEFINED;}
\text{if} \ d == 31 || s == 31 || n == 31 \ \text{then UNDEFINED;}

Assemblers Symbols

\texttt{<Xd>} \hspace{1em} \text{For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.}
\hspace{1em} \text{For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.}

\texttt{<Xs>} \hspace{1em} \text{For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.}
\hspace{1em} \text{For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.}

\texttt{<Xn>} \hspace{1em} \text{For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.}
\hspace{1em} \text{For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.}
\hspace{1em} \text{For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.}
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;

bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

        if SInt(cpysize) > 0 then
            assert SInt(stagecpysize) <= SInt(cpysize);
        else
            assert SInt(stagecpysize) >= SInt(cpysize);
    
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

                readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
                Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
                fromaddress = fromaddress + B;
toadress = toadress + B;

                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;

                if stage != MOPSSStage_Prologue then
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;

                if stage == MOPSSStage_Prologue then
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Memory Copy Forward-only, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTRN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
-Xd holds the original Xd + saturated Xn.
-Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
-PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTRN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
-Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
-Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
-Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
-PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
-Xd holds the lowest address that the copy is made to -Xn.
-At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
-Xd holds the lowest address that the copy is made to.
-At the end of the instruction:
  ◦ the value of Xn is written back with the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  ◦ the value of Xs is written back with the lowest address that has not been copied from.
  ◦ the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
-Xd holds the lowest address that the copy is made to -Xn.
-At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
-Xd holds the lowest address that the copy is made to.
-At the end of the instruction:
  ◦ the value of Xn is written back with 0.
the value of Xs is written back with the lowest address that has not been copied from.
the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FeaT_MOPS)

| sz | 0 | 1 | 1 | 0 | 0 | 1 | op1 | 0 | Rs | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd |

Epilogue (op1 == 10)

CPYFEWTRN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYFMWTRN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYFPWTRN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

  if supports_option_a then
    PSTATE.C = '0';
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';

  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);

else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if PSTATE.C == '1' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if PSTATE.C == '0' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSSStage_Main then
      stagecpysize = cpysize - postsize;

      // Check if the parameters to this instruction are valid.
      if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        stagecpysize = postsize;

      // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n] = cpysize;
        end if
    end while
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n] = cpysize;
            X[d] = toaddress;
            X[s] = fromaddress;
        end if
    end while
end if

if stage == MOPSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
end if
CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory Copy Forward-only, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN. CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

After execution of CPYFPWTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.(N,Z,V) are set to {0,0,0}.

For CPYFMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
- the value of $X_s$ is written back with the lowest address that has not been copied from.
- the value of $X_d$ is written back with the lowest address that has not been copied to.

**Integer**

(\texttt{FEAT\_MOPS})

| sz | 0 | 1 | 1 | 0 | 0 | 1 | op1 | 0 | Rs | 0 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
|----|---|---|---|---|---|---|-----|---|----|---|---|---|---|---|----|---|

**Epilogue (op1 == 10)**

\texttt{CPYFEWTWN} \[, [<X_d>]!, [<X_s>]!, <X_n>!\]

**Main (op1 == 01)**

\texttt{CPYFMWTWN} \[, [<X_d>]!, [<X_s>]!, <X_n>!\]

**Prologue (op1 == 00)**

\texttt{CPYFPWTWN} \[, [<X_d>]!, [<X_s>]!, <X_n>!\]

if \(!\texttt{HaveFeatMOPS}()\) then UNDEFINED;
if sz \(!= \texttt{00}\) then UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer s = \texttt{UInt}(Rs);
integer n = \texttt{UInt}(Rn);
bits(4) options = op2;

\texttt{MOPSSStage} stage;
\texttt{case} op1 of
  when \texttt{00} \texttt{stage} = \texttt{MOPSSStage\_Prologue};
  when \texttt{01} \\
  when \texttt{10} stage = \texttt{MOPSSStage\_Epilogue};
  otherwise \texttt{SEE "Memory Copy and Memory Set"};
\texttt{if} d == s || s == n || d == n then UNDEFINED;
\texttt{if} d == 31 || s == 31 || n == 31 then UNDEFINED;

**Assembler Symbols**

\texttt{<X_d>}

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

\texttt{<X_s>}

For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

\texttt{<X_n>}

For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63> == '1' then cpysize = 0xFFFFFFFFFFFFFFFF<63:0>;
    if supports_option_a then
      PSTATE.C = '0';
      // Copy in the forward direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = Zeros(64) - cpysize;
    else
      PSTATE.C = '1';
      PSTATE.N = '0';
      PSTATE.V = '0';
      PSTATE.Z = '0';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
      assert SInt(stagecpysize) <= SInt(cpysize);
    else
      assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
      if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;
        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
          boolean wrong_option = FALSE;
          boolean from_epilogue = FALSE;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          stagecpysize = postsize;
          // Check if the parameters to this instruction are valid for the epilogue.
if (cpysize != postsize || MemCpyParametersIllformed(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
        end
    end
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        if stage != MOPSSStage_Prologue then
            X[n] = cpysize;
            X[d] = toaddress;
            X[s] = fromaddress;
        end
    end
endif

Memory Copy. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE. CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYE performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYP, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.
After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elseif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.
The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYP, option A (which results in encoding PSTATE.C = 0):
• PSTATE.{N,Z,V} are set to {0,0,0}.
  • If the copy is in the forward direction, then:
    ◦ Xs holds the original Xs + saturated Xn.
    ◦ Xd holds the original Xd + saturated Xn.
    ◦ Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
  • If the copy is in the backward direction, then:
    ◦ Xs and Xd are unchanged.
    ◦ Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYP, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  ◦ Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  ◦ Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYM, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is copied to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYM, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
• Xs holds the lowest address that the copy is copied from.
• Xd holds the lowest address that the copy is copied to.
• At the end of the instruction:
  ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  ■ the value of Xs is written back with the lowest address that has not been copied from.
  ■ the value of Xd is written back with the lowest address that has not been copied to.

  • If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
      ■ the value of Xs is written back with the highest address that has not been copied from +1.
      ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYE, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is made to.
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYE, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from.
  ◦ Xd holds the highest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 0  | 1  | 1  | 1  | 0  | 1  | op1| 0  | Rs | 0  | 0  | 0  | 0  | 0  | 0  | 1  | Rn | Rd |

op2
Epilogue (op1 == 10)

CPYE [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYM [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYP [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) &
      (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &
      forward = TRUE;
  elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &
      (UInt(fromaddress<55:0>) + cpysize<55:0>) >
      UInt(toaddress<55:0>)) then
    forward = FALSE;
  else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
  if supports_option_a then
    PSTATE.C = '0';
    PSTATE.N = '0';
    if forward then
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = Zeros(64) - cpysize;
    else
      PSTATE.C = '1';
      PSTATE.N = '0';
      PSTATE.V = '0';
      PSTATE.Z = '0';
      // IMP DEF selection of the amount covered by pre-processing.
      stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
      assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
      if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
      else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
      boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        if PSTATE.C == '0' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      }
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    if supports_option_a then
      while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
          assert B <= -1 * SInt(stagecpysize);
          readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
          Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
          cpysize = cpysize + B;
          stagecpysize = stagecpysize + B;
        else
          assert B <= SInt(stagecpysize);
          cpysize = cpysize - B;
          stagecpysize = stagecpysize - B;
          readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
          Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
  else
    while UINT(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= UINT(stagecpysize);

      if PSTATE.N == '0' then
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;

      if stage != MOPSStage_Prologue then
        X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
if stage == `MOPSStage_Prologue` then
    $X[n] = cpysize$;
    $X[d] = toaddress$;
    $X[s] = fromaddress$;
CPYPN, CPYMN, CPYEN

Memory Copy, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPN, the following saturation logic is applied:
If \( Xn < 63:55 \neq 000000000 \), the copy size \( Xn \) is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
- If \( (Xs > Xd) \&\& (Xd + saturated Xn) > Xs \), then direction = forward
- Elsif \( (Xs < Xd) \&\& (Xs + saturated Xn) > Xd \), then direction = backward
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPN, option A (which results in encoding PSTATE.C = 0):
- PSTATE.{N,Z,V} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - \( Xs \) holds the original \( Xs + saturated Xn \).
  - \( Xd \) holds the original \( Xd + saturated Xn \).
  - \( Xn \) holds -1* saturated \( Xn \) + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - \( Xs \) and \( Xd \) are unchanged.
  - \( Xn \) holds the saturated value of \( Xn \) - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPN, option B (which results in encoding PSTATE.C = 1):
- If the copy is in the forward direction, then:
  - \( Xs \) holds the original \( Xs + an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( Xd \) holds the original \( Xd + an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( Xn \) holds the saturated \( Xn - an IMPLEMENTATION DEFINED number of bytes copied \).
  - PSTATE.{N,Z,V} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - \( Xs \) holds the original \( Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( Xd \) holds the original \( Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied \).
  - \( Xn \) holds the saturated \( Xn - an IMPLEMENTATION DEFINED number of bytes copied \).
  - PSTATE.{N,Z,V} are set to \{1,0,0\}.

For CPYMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- \( Xn \) is treated as a signed 64-bit number.
- If the copy is in the forward direction (\( Xn \) is a negative number), then:
  - \( Xn \) holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - \( Xs \) holds the lowest address that the copy is copied from \(-Xn\).
  - \( Xd \) holds the lowest address that the copy is made to \(-Xn\).
  - At the end of the instruction, the value of \( Xn \) is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (\( Xn \) is a positive number), then:
  - \( Xn \) holds the number of bytes remaining to be copied in the memory copy in total.
  - \( Xs \) holds the highest address that the copy is copied from \(-Xn+1\).
  - \( Xd \) holds the highest address that the copy is copied to \(-Xn+1\).
  - At the end of the instruction, the value of \( Xn \) is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- \( Xn \) holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
  • If the copy is in the forward direction (Xn is a negative number), then:
    ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the lowest address that the copy is copied from -Xn.
    ◦ Xd holds the lowest address that the copy is made to -Xn.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
  • If the copy is in the backward direction (Xn is a positive number), then:
    ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the highest address that the copy is copied from -Xn+1.
    ◦ Xd holds the highest address that the copy is copied to -Xn+1
    ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
  • If the copy is in the forward direction (PSTATE.N == 0), then:
    ◦ Xs holds the lowest address that the copy is copied from.
    ◦ Xd holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the lowest address that has not been copied from.
      ■ the value of Xd is written back with the lowest address that has not been copied to.
  • If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the highest address that has not been copied from +1.
      ■ the value of Xd is written back with the highest address that has not been copied to +1.

**Integer**

(*FEAT_MOPS*)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td>0</td>
<td>Rs</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| op2 |
Epilogue (op1 == 10)

CPYEN [<Xd>], [<Xs>], <Xn>

Main (op1 == 01)

CPYMN [<Xd>], [<Xs>], <Xn>

Prologue (op1 == 00)

CPYPN [<Xd>], [<Xs>], <Xn>

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;
case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) + cpysize<55:0>) then
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) + cpysize<55:0>) then
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            PSTATE.N = '0';
        end if;

    else
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    end if;

    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            end if;
        else
            if PSTATE.C == '0' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            end if;
        end if;
    end if;
end if;
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;
    // Check if the parameters to this instruction are valid.
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;
        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    if supports_option_a then
        while SInt(stagecpysize) != 0 do
            // IMP DEF selection of the block size that is worked on. While many
            // implementations might make this constant, that is not assumed.
            B = CPYSizeChoice(toaddress, fromaddress, cpysize);

            if SInt(cpysize) < 0 then
                assert B <= -1 * SInt(stagecpysize);
                readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                cpysize = cpysize + B;
                stagecpysize = stagecpysize + B;
            else
                assert B <= SInt(stagecpysize);
                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;
                readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        if stage != MOPSStage_Prologue then
            X[n] = cpysize;
        else
            while UInt(stagecpysize) > 0 do
                // IMP DEF selection of the block size that is worked on. While many
                // implementations might make this constant, that is not assumed.
                B = CPYSizeChoice(toaddress, fromaddress, cpysize);
                assert B <= UInt(stagecpysize);

                if PSTATE.N == '0' then
                    readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
                    Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress + B;
                    toaddress = toaddress + B;
                else
                    readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
                    Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
                    fromaddress = fromaddress - B;
                    toaddress = toaddress - B;
                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;

                if stage != MOPSStage_Prologue then
                    X[n] = cpysize;
                    X[d] = toaddress;
                    X[s] = fromaddress;
                    break;
if stage == MOPSStage_Prologue then
    \[ X[n] = \text{cpysize}; \]
    \[ X[d] = \text{toaddress}; \]
    \[ X[s] = \text{fromaddress}; \]
CPYPRN, CPYMRN, CPYERN

Memory Copy, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elseif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds \(-1\times\) the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from \(-Xn\).
  ◦ Xd holds the lowest address that the copy is made to \(-Xn\).
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from \(-Xn+1\).
  ◦ Xd holds the highest address that the copy is copied to \(-Xn+1\).
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(\texttt{FEAT}\_\texttt{MOPS})

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sz | 0 1 1 1 | 0 1 | op1 | 0 | Rs | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd
op2
\end{verbatim}
Epilogue (op1 == 10)

CPYERN [Xd], [Xs], Xn!

Main (op1 == 01)

CPYMRN [Xd], [Xs], Xn!

Prologue (op1 == 00)

CPYPRN [Xd], [Xs], Xn!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) then
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) then
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
            else
                PSTATE.N = '0';
                PSTATE.Z = '0';
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
```

bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
    stagecpysize = cpysize - postsize;

    // Check if the parameters to this instruction are valid.
    if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = FALSE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

    // Check if the parameters to the epilogue are valid.
    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
        boolean wrong_option = FALSE;
        boolean from_epilogue = TRUE;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

else
    stagecpysize = cpysize;

    if supports_option_a then
        while SInt(stagecpysize) != 0 do

            B = CPYSizeChoice(toaddress, fromaddress, cpysize);

            if SInt(cpysize) < 0 then
                assert B <= -1 * SInt(stagecpysize);
                readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
                cpysize = cpysize + B;
                stagecpysize = stagecpysize + B;
            else
                assert B <= SInt(stagecpysize);
                cpysize = cpysize - B;
                stagecpysize = stagecpysize - B;
                readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
                Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

            if stage != MOPSStage_Prologue then
                X[n] = cpysize;
            end
        end

    else
        while UINT(stagecpysize) > 0 do

            B = CPYSizeChoice(toaddress, fromaddress, cpysize);

            if PSTATE.N == '0' then
                readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
                Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
                fromaddress = fromaddress + B;
                toaddress = toaddress + B;
            else
                readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
                Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
                fromaddress = fromaddress - B;
                toaddress = toaddress - B;

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            if stage != MOPSStage_Prologue then
                X[n] = cpysize;
                X[d] = toaddress;
                X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    $X[n] = cpysize$
    $X[d] = toaddress$
    $X[s] = fromaddress$
Memory Copy, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRT, the following saturation logic is applied:
If \( X_n < 63:55 \) \(!= 0000000000 \), the copy size \( X_n \) is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If \( (X_s > X_d) \) \&\& \( (X_d + \text{saturated } X_n) > X_s \), then direction = forward
Elsif \( (X_s < X_d) \) \&\& \( (X_s + \text{saturated } X_n) > X_d \), then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRT, option A (which results in encoding PSTATE.C = 0):
• PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
  • If the copy is in the forward direction, then:
    ◦ \( X_s \) holds the original \( X_s + \text{saturated } X_n \).
    ◦ \( X_d \) holds the original \( X_d + \text{saturated } X_n \).
    ◦ \( X_n \) holds -1* \( \text{saturated } X_n \) + an IMPLEMENTATION DEFINED number of bytes copied.
  • If the copy is in the backward direction, then:
    ◦ \( X_s \) and \( X_d \) are unchanged.
    ◦ \( X_n \) holds the saturated value of \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRT, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  ◦ \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  ◦ \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  ◦ \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
• If the copy is in the backward direction, then:
  ◦ \( X_s \) holds the original \( X_s + \text{saturated } X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ \( X_d \) holds the original \( X_d + \text{saturated } X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.\{N,Z,V\} are set to \{1,0,0\}.

For CPYMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• \( X_n \) is treated as a signed 64-bit number.
• If the copy is in the forward direction (\( X_n \) is a negative number), then:
  ◦ \( X_n \) holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ \( X_s \) holds the lowest address that the copy is copied from -\( X_n \).
  ◦ \( X_d \) holds the lowest address that the copy is made to -\( X_n \).
  ◦ At the end of the instruction, the value of \( X_n \) is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
• If the copy is in the backward direction (\( X_n \) is a positive number), then:
  ◦ \( X_n \) holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ \( X_s \) holds the highest address that the copy is copied from -\( X_n + 1 \).
  ◦ \( X_d \) holds the highest address that the copy is copied to -\( X_n + 1 \).
  ◦ At the end of the instruction, the value of \( X_n \) is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• \( X_n \) holds the number of bytes to be copied in the memory copy in total.
If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.

If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
  ◦ Xn is treated as a signed 64-bit number.
  If the copy is in the forward direction (Xn is a negative number), then:
    ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the lowest address that the copy is copied from -Xn.
    ◦ Xd holds the lowest address that the copy is made to -Xn.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
  If the copy is in the backward direction (Xn is a positive number), then:
    ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the highest address that the copy is copied from -Xn+1.
    ◦ Xd holds the highest address that the copy is copied to -Xn+1.
    ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
  ◦ Xn holds the number of bytes to be copied in the memory copy in total
  If the copy is in the forward direction (PSTATE.N == 0), then:
    ◦ Xs holds the lowest address that the copy is copied from.
    ◦ Xd holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      ▪ the value of Xn is written back with 0.
      ▪ the value of Xs is written back with the lowest address that has not been copied from.
      ▪ the value of Xd is written back with the lowest address that has not been copied to.
  If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ▪ the value of Xn is written back with 0.
      ▪ the value of Xs is written back with the highest address that has not been copied from +1.
      ▪ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPYRT, CPYMRT, CPYERT
Epilogue (op1 == 10)

CPYERT [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMRT [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPRT [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>))
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
    if supports_option_a then
        PSTATE.C = '0';
PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                PSTATE.N = '1';
            else
                PSTATE.N = '0';
PSTATE.V = '0';
PSTATE.Z = '0';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();
        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

            CPYPRT, CPYMRT, CPYERT
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;
  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;
    // Check if the parameters to the epilogue are valid.
    if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
      boolean wrong_option = FALSE;
      boolean from_epilogue = TRUE;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  fi
fi

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    fi
  fi
fi

if stage != MOPSStage_Prologue then
  X[n] = cpysize;
else
  while UINT(stagecpysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UINT(stagecpysize);
    if PSTATE.N == '0' then
      readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
      Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;
    else
      readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
      Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress - B;
      toaddress = toaddress - B;
    fi
    cpysize = cpysize - B;
    stagecpysize = stagecpysize - B;
  fi
fi

if stage != MOPSStage_Prologue then
  X[n] = cpysize;
  X[d] = toaddress;
  X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    $X[n] = cpysize;$
    $X[d] = toaddress;$
    $X[s] = fromaddress;$
Memory Copy, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTN performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTN, the following saturation logic is applied:

If \( X_n < 63:55 \) != 0000000000, the copy size \( X_n \) is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

- If \((X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s\), then direction = forward
- Elsif \((X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d\), then direction = backward
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated } X_n \).
  - \( X_d \) holds the original \( X_d + \text{saturated } X_n \).
  - \( X_n \) holds \(-1^* \text{saturated } X_n + \text{ an IMPLEMENTATION DEFINED number of bytes copied.}\)
- If the copy is in the backward direction, then:
  - \( X_s \) and \( X_d \) are unchanged.
  - \( X_n \) holds the saturated value of \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{ an IMPLEMENTATION DEFINED number of bytes copied.}\)
  - \( X_d \) holds the original \( X_d + \text{ an IMPLEMENTATION DEFINED number of bytes copied.}\)
  - \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated } X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  - \( X_d \) holds the original \( X_d + \text{saturated } X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  - \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.\{N,Z,V\} are set to \{1,0,0\}.

For CPYMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- \( X_n \) is treated as a signed 64-bit number.
- If the copy is in the forward direction (\( X_n \) is a negative number), then:
  - \( X_n \) holds \(-1^*\) the number of bytes remaining to be copied in the memory copy in total.
  - \( X_s \) holds the lowest address that the copy is copied from \(-X_n\).
  - \( X_d \) holds the lowest address that the copy is made to \(-X_n\).
  - At the end of the instruction, the value of \( X_n \) is written back with \(-1^*\) the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (\( X_n \) is a positive number), then:
  - \( X_n \) holds the number of bytes remaining to be copied in the memory copy in total.
  - \( X_s \) holds the highest address that the copy is copied from \(-X_n+1\).
  - \( X_d \) holds the highest address that the copy is copied to \(-X_n+1\).
  - At the end of the instruction, the value of \( X_n \) is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- \( X_n \) holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is made to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

**Integer (FEAT_MOPS)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 1  | 0  | 1  | op1| 0  |    | Rs | 1  | 1  | 1  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op2|

CPYPRTN, CPYMRTN, CPYERTN
Epilogue (op1 == 10)

    CPYERTN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

    CPYMRTN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

    CPYPRTN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

    integer d = UInt(Rd);
    integer s = UInt(Rs);
    integer n = UInt(Rn);
    bits(4) options = op2;

    MOPSSStage stage;
    case op1 of
        when '00' stage = MOPSSStage_Prologue;
        when '01' stage = MOPSSStage_Main;
        when '10' stage = MOPSSStage_Epilogue;
        otherwise SEE "Memory Copy and Memory Set";
    end

    if d == s || s == n || d == n then UNDEFINED;
    if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);
boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);
if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if (UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) then
        forward = TRUE;
    elsif (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) < UInt(toaddress<55:0>)) then
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                PSTATE.N = '1';
            else
                PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
    else
        boolean zero_size_exceptions = MemCpyZeroSizeCheck();
        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(cpysize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSStage_Epilogue;
                        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    CPYPRTN, CPYMRTN, CPYERTN
bits(64) postsize = \text{CPYPostSizeChoice}(toaddress, fromaddress, cpysize);
assert postsize\leq63 \implies cpysize\leq63 \lor SInt(postsize) = 0;

if stage == \text{MOPSStage\_Main} then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize \or\! \text{MemCpyParametersIllformedE}(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  endif

if supports_option_a then
  while SInt(stagecpysize) \neq 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = \text{CPYSizeChoice}(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 \times SInt(stagecpysize);
      readdata<8*1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<8*1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<8*1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<8*1:0>;
    endif

  if stage != \text{MOPSStage\_Prologue} then
    X[n] = cpysize;
  else
    while UInt(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = \text{CPYSizeChoice}(toaddress, fromaddress, cpysize);
      assert B <= UInt(stagecpysize);

      if PSTATE.N == 0 then
        readdata<8*1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<8*1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<8*1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<8*1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;
      endif

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;

    if stage != \text{MOPSStage\_Prologue} then
      X[n] = cpysize;
      X[d] = toaddress;
      X[s] = fromaddress;
    endif
  endif

if stage == MOPSStage_Prologue then
  \( X[n] = \text{cpysize}; \)
  \( X[d] = \text{toaddress}; \)
  \( X[s] = \text{fromaddress}; \)
Memory Copy, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and performs an implementation-defined amount of the memory copy. CPYMRTRN performs an implementation-defined amount of the memory copy. CPYERTRN performs the last part of the memory copy.

**Note**

The inclusion of implementation-defined amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTRN, the following saturation logic is applied:
If Xn\text{<}_63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elseif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = implementation-defined choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation-defined.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTRN, option A (which results in encoding PSTATE.C = 0):
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an implementation-defined number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an implementation-defined number of bytes copied.

After execution of CPYPRTRN, option B (which results in encoding PSTATE.C = 1):
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an implementation-defined number of bytes copied.
  - Xd holds the original Xd + an implementation-defined number of bytes copied.
  - Xn holds the saturated Xn - an implementation-defined number of bytes copied.
  - PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an implementation-defined number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an implementation-defined number of bytes copied.
  - Xn holds the saturated Xn - an implementation-defined number of bytes copied.
  - PSTATE.\{N,Z,V\} are set to \{1,0,0\}.

For CPYMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
  - If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
  - If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is made to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is made to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYERTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with 0.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)
Epilogue (op1 == 10)

CPYERTRN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMRTRN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPRTRN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
    case op1 of
        when '00' stage = MOPSSStage_Prologue;
        when '01' stage = MOPSSStage_Main;
        when '10' stage = MOPSSStage_Epilogue;
        otherwise SEE "Memory Copy and Memory Set";
    end case;

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
integer N = MaxBlockSizeCopiedBytes();

bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
        boolean forward;
        if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) + cpysize<55:0> + cpysize<55:0>)
            forward = TRUE;
        elseif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) + cpysize<55:0>)
            forward = FALSE;
        else
            forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                PSTATE.N = '1';
            else
                PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;
if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;
  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;
  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
    else
      while Uint(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= Uint(stagecpysize);
        if PSTATE.N == ‘0' then
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else
          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;
        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
      if stage != MOPSStage_Prologue then
        X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    \[X[n] = cpysize;\]
    \[X[d] = toaddress;\]
    \[X[s] = fromaddress;\]
Memory Copy, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTWN performs the last part of the memory copy.

Note
The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTWN, the following saturation logic is applied:
If \( X_n < 63:55 \) != 0000000000, the copy size \( X_n \) is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If \( (X_s > X_d) \) && \( (X_d + \text{saturated } X_n) > X_s \), then direction = forward
Elsif \( (X_s < X_d) \) && \( (X_s + \text{saturated } X_n) > X_d \), then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note
Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTWN, option A (which results in encoding PSTATE.C = 0):
• PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the forward direction, then:
  ◦ \( X_s \) holds the original \( X_s + \text{saturated } X_n \).
  ◦ \( X_d \) holds the original \( X_d + \text{saturated } X_n \).
  ◦ \( X_n \) holds \(-1\times \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
• If the copy is in the backward direction, then:
  ◦ \( X_s \) and \( X_d \) are unchanged.
  ◦ \( X_n \) holds the saturated value of \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTWN, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  ◦ \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
  ◦ \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
  ◦ \( X_n \) holds the saturated \( X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
  ◦ PSTATE.{N,Z,V} are set to \{0,0,0\}.
• If the copy is in the backward direction, then:
  ◦ \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
  ◦ \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
  ◦ \( X_n \) holds the saturated \( X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied.}\)
  ◦ PSTATE.{N,Z,V} are set to \{1,0,0\}.

For CPYMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• \( X_n \) is treated as a signed 64-bit number.
• If the copy is in the forward direction (\( X_n \) is a negative number), then:
  ◦ \( X_n \) holds \(-1\times \text{the number of bytes remaining to be copied in the memory copy in total.}\)
  ◦ \( X_s \) holds the lowest address that the copy is copied from -\( X_n \).
  ◦ \( X_d \) holds the lowest address that the copy is made to -\( X_n \).
  ◦ At the end of the instruction, the value of \( X_n \) is written back with \(-1\times \text{the number of bytes remaining to be copied in the memory copy in total.}\)
• If the copy is in the backward direction (\( X_n \) is a positive number), then:
  ◦ \( X_n \) holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ \( X_s \) holds the highest address that the copy is copied from -\( X_n +1 \).
  ◦ \( X_d \) holds the highest address that the copy is copied to -\( X_n +1 \).
  ◦ At the end of the instruction, the value of \( X_n \) is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• \( X_n \) holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
  • If the copy is in the forward direction (Xn is a negative number), then:
    ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the lowest address that the copy is copied from -Xn.
    ◦ Xd holds the lowest address that the copy is made to -Xn.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
  • If the copy is in the backward direction (Xn is a positive number), then:
    ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the highest address that the copy is copied from -Xn+1.
    ◦ Xd holds the highest address that the copy is copied to -Xn+1.
    ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total.
  • If the copy is in the forward direction (PSTATE.N == 0), then:
    ◦ Xs holds the lowest address that the copy is copied from.
    ◦ Xd holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the lowest address that has not been copied from.
      ■ the value of Xd is written back with the lowest address that has not been copied to.
  • If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the highest address that has not been copied from +1.
      ■ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

|   |  31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op2 | 0 | 1 | 1 | 1 | 0 | 1 | op1 | 0 |
| Rs  | 0 | 1 | 1 | 0 | 0 | 1 | Rn  |   |
| Rd  |   |   |   |   |   |   |     |   |

CPYPRTWN, CPYMRTWN, CPYERTWN
Epilogue (op1 == 10)

CPYERTWN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMRTWN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPRTWN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

XS> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>))<true>)
    forward = TRUE;
  elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>))
    forward = FALSE;
  else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
  if supports_option_a then
    PSTATE.C = '0';
    PSTATE.N = '0';
    if forward then
      // Copy in the forward direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = Zeros(64) - cpysize;
    else
      PSTATE.C = '1';
      if !forward then
        // Copy in the reverse direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
      PSTATE.N = '1';
    else
      PSTATE.N = '0';
      PSTATE.V = '0';
      PSTATE.Z = '0';
      // IMP DEF selection of the amount covered by pre-processing.
      stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
      assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
      assert SInt(stagecpysize) <= SInt(cpysize);
    else
      assert SInt(stagecpysize) >= SInt(cpysize);
    else
      boolean zero_size_exceptions = MemCpyZeroSizeCheck();
      // Check if this version is consistent with the state of the call.
      if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
          if PSTATE.C == '1' then
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if PSTATE.C == '0' then
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if PSTATE.N == '0' then
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else
          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
          X[d] = toaddress;
          X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    \( X[n] = \text{cpysize} \);
    \( X[d] = \text{toaddress} \);
    \( X[s] = \text{fromaddress} \);
CPYPT, CPYMT, CPYET

Memory Copy, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYET performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPT, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPT, option A (which results in encoding PSTATE.C = 0):
• PSTATE.{N,Z,V} are set to {0,0,0}.
  • If the copy is in the forward direction, then:
    o Xs holds the original Xs + saturated Xn.
    o Xd holds the original Xd + saturated Xn.
    o Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
  • If the copy is in the backward direction, then:
    o Xs and Xd are unchanged.
    o Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPT, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  o Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  o Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  o Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  o PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  o Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  o Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  o Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  o PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
  • If the copy is in the forward direction (Xn is a negative number), then:
    o Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    o Xs holds the lowest address that the copy is copied from -Xn.
    o Xd holds the lowest address that the copy is made to -Xn.
    o At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
  • If the copy is in the backward direction (Xn is a positive number), then:
    o Xn holds the number of bytes remaining to be copied in the memory copy in total.
    o Xs holds the highest address that the copy is copied from -Xn+1.
    o Xd holds the highest address that the copy is copied to -Xn+1.
    o At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total.
If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.

If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYET, option A (encoded by PSTATE.C = 0), the format of the arguments is:
  • Xn is treated as a signed 64-bit number.
  • If the copy is in the forward direction (Xn is a negative number), then:
    ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the lowest address that the copy is copied from -Xn.
    ◦ Xd holds the lowest address that the copy is copied to -Xn.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
  • If the copy is in the backward direction (Xn is a positive number), then:
    ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the highest address that the copy is copied from -Xn+1.
    ◦ Xd holds the highest address that the copy is copied to -Xn+1.
    ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYET, option B (encoded by PSTATE.C = 1), the format of the arguments is:
  • Xn holds the number of bytes to be copied in the memory copy in total
  • If the copy is in the forward direction (PSTATE.N == 0), then:
    ◦ Xs holds the lowest address that the copy is copied from.
    ◦ Xd holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the lowest address that has not been copied from.
      ■ the value of Xd is written back with the lowest address that has not been copied to.
  • If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the highest address that has not been copied from +1.
      ■ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Epilogue (op1 == 10)

CPYET \[<Xd>\], \[<Xs>\], \[<Xn>\]

Main (op1 == 01)

CPYMT \[<Xd>\], \[<Xs>\], \[<Xn>\]

Prologue (op1 == 00)

CPYPT \[<Xd>\], \[<Xs>\], \[<Xn>\]

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSStage stage;

\[
\begin{align*}
\text{case op1 of} \\
&\text{when '00' stage = MOPSStage Prologue;} \\
&\text{when '01' stage = MOPSStage Main;} \\
&\text{when '10' stage = MOPSStage Epilogue;} \\
&\text{otherwise SEE "Memory Copy and Memory Set";} \\
\end{align*}
\]

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = \texttt{MaxBlockSizeCopiedBytes}();

bits(64) toaddress = \texttt{X[d]};
bits(64) fromaddress = \texttt{X[s]};
bits(64) cpysize = \texttt{X[n]};
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if \texttt{HaveMTE2Ext}() then
  \texttt{SetTagCheckedInstruction(TRUE)};

boolean supports\_option\_a = \texttt{MemCpyOptionA}();
(racctype, wacctype) = \texttt{MemCpyAccessTypes(options)};

if stage == \texttt{MOPSSStage\_Prologue} then
  if \texttt{cpysize<63:55>} != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

  boolean forward;
  if ((\texttt{UInt(fromaddress<55:0>)} > \texttt{UInt(toaddress<55:0>)})) && (\texttt{UInt(fromaddress<55:0>)} < \texttt{UInt(toaddress<55:0> + cpysize<55:0>)})
    forward = TRUE;
  elsif ((\texttt{UInt(fromaddress<55:0>)} < \texttt{UInt(toaddress<55:0>)})) && (\texttt{UInt(fromaddress<55:0> + cpysize<55:0>)})
    forward = FALSE;
  else
    forward = \texttt{MemCpyDirectionChoice(fromaddress, toaddress, cpysize)};

  if supports\_option\_a then
    PSTATE.C = '0';
    PSTATE.N = '0';
    if forward then
      // Copy in the forward direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = \texttt{Zeros(64)} - cpysize;
    else
      PSTATE.C = '1';
      if !forward then
        // Copy in the reverse direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        PSTATE.N = '1';
      else
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = \texttt{CPYPreSizeChoice(toaddress, fromaddress, cpysize)};
        assert stagecpysize<63> == \texttt{cpysize<63>} || stagecpysize == \texttt{Zeros()};

        if \texttt{SInt(cpysize)} > 0 then
          assert \texttt{SInt(stagecpysize)} <= \texttt{SInt(cpysize)};
        else
          assert \texttt{SInt(stagecpysize)} >= \texttt{SInt(cpysize)};
        end;
    end;
  end;

  boolean zero\_size\_exceptions = \texttt{MemCpyZeroSizeCheck}();

  // Check if this version is consistent with the state of the call.
  if zero\_size\_exceptions || \texttt{SInt(cpysize)} != 0 then
    if supports\_option\_a then
      if PSTATE.C == '1' then
        boolean wrong\_option = TRUE;
        boolean from\_epilogue = stage == \texttt{MOPSStage\_Epilogue};
        \texttt{MismatchedMemCpyException(supports\_option\_a, d, s, n, wrong\_option, from\_epilogue, options)};
      else
        if PSTATE.C == '0' then
          boolean wrong\_option = TRUE;
          boolean from\_epilogue = stage == \texttt{MOPSStage\_Epilogue};
          \texttt{MismatchedMemCpyException(supports\_option\_a, d, s, n, wrong\_option, from\_epilogue, options)};
        end;
      end;
    end;
  end;
end;
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

  if stage != MOPSStage_Prologue then
    X[n] = cpysize;
  else
    while UInt(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= UInt(stagecpysize);

      if PSTATE.N == '0' then
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;

      if stage != MOPSStage_Prologue then
        X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
        CPYPT, CPYMT, CPYET
if stage == MOPSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
CPYPTN, CPYMTN, CPYETN

Memory Copy, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETN performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTN, the following saturation logic is applied:

If Xn<63:55> != 0000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is copied to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with 0.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with 0.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 1  | 0  | 1  | op1| 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | Rs |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op2|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Epilogue (op1 == 10)

CPYETN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMTN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPTN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
MOPSSStage stage;
  case op1 of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";
  end;
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
integer N = MaxBlockSizeCopiedBytes();

bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

boolean forward;
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) &
    (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &
    forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &
    (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>)) &
    forward = FALSE;
else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports option a then
    PSTATE.C = '0';
PSTATE.N = '0';
if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
else
    PSTATE.C = '1';
if !forward then
    // Copy in the reverse direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
PSTATE.N = '1';
else
    PSTATE.N = '0';
PSTATE.V = '0';
PSTATE.Z = '0';

// IMP DEF selection of the amount covered by pre-processing.
stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports option a then
        if PSTATE.C == '1' then
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
            if PSTATE.C == '0' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformed(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, racctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, racctype] = readdata<B*8-1:0>;

    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if PSTATE.N == '0' then
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, racctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else
          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, racctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
          X[d] = toaddress;
          X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
**CPYPTRN, CPYMTRN, CPYETRN**

Memory Copy, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETRN performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

- If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
- Elsid (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is made to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
  • If the copy is in the forward direction (Xn is a negative number), then:
    ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the lowest address that the copy is copied from -Xn.
    ◦ Xd holds the lowest address that the copy is made to -Xn.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
  • If the copy is in the backward direction (Xn is a positive number), then:
    ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the highest address that the copy is copied from -Xn+1.
    ◦ Xd holds the highest address that the copy is copied to -Xn+1.
    ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
  • If the copy is in the forward direction (PSTATE.N == 0), then:
    ◦ Xs holds the lowest address that the copy is copied from.
    ◦ Xd holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      ▪ the value of Xn is written back with 0.
      ▪ the value of Xs is written back with the lowest address that has not been copied from.
      ▪ the value of Xd is written back with the lowest address that has not been copied to.
  • If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ▪ the value of Xn is written back with 0.
      ▪ the value of Xs is written back with the highest address that has not been copied from +1.
      ▪ the value of Xd is written back with the highest address that has not been copied to +1.

**Integer (FEAT_MOPS)**

<table>
<thead>
<tr>
<th></th>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Epilogue (op1 == 10)

CPYETRN [<Xd>], [<XS>], <Xn>!

Main (op1 == 01)

CPYMTRN [<Xd>], [<XS>], <Xn>!

Prologue (op1 == 00)

CPYPTRN [<Xd>], [<XS>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;

case op1 of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
  if supports option a then
    PSTATE.C = '0';
    PSTATE.N = '0';
    if forward then
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = Zeros(64) - cpysize;
    else
      PSTATE.C = '1';
      if !forward then
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        PSTATE.N = '1';
      else
        PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';
  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();
  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports option a then
      if PSTATE.C == '1' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        if PSTATE.C == '0' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
          else
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  if supports_option_a then
    while SInt(stagecpysize) != 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);

      if SInt(cpysize) < 0 then
        assert B <= -1 * SInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
      else
        assert B <= SInt(stagecpysize);

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

      if stage != MOPSStage_Prologue then
        X[n] = cpysize;
      else
        while UInt(stagecpysize) > 0 do
          // IMP DEF selection of the block size that is worked on. While many
          // implementations might make this constant, that is not assumed.
          B = CPYSizeChoice(toaddress, fromaddress, cpysize);
          assert B <= UInt(stagecpysize);

          if PSTATE.N == '0' then
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
          else
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

          cpysize = cpysize - B;
          stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
      end if
    end while
  end if
else
  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  if supports_option_a then
    while UInt(stagecpysize) != 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);

      if UInt(cpysize) < 0 then
        assert B <= -1 * UInt(stagecpysize);
        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;
      else
        assert B <= UInt(stagecpysize);

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;
        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

      if stage != MOPSStage_Prologue then
        X[n] = cpysize;
      else
        while SInt(stagecpysize) > 0 do
          // IMP DEF selection of the block size that is worked on. While many
          // implementations might make this constant, that is not assumed.
          B = CPYSizeChoice(toaddress, fromaddress, cpysize);
          assert B <= SInt(stagecpysize);

          if PSTATE.N == '0' then
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
          else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

          cpysize = cpysize - B;
          stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
      end if
    end while
  end if
end if
if stage == MOPSStage_Prologue then
  X[n] = cpysize;
  X[d] = toaddress;
  X[s] = fromaddress;

Memory Copy, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTWN, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
ElseIf (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is made to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

For CPYETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
If the copy is in the forward direction (PSTATE.N == 0), then:
- $X_s$ holds the lowest address that the copy is copied from.
- $X_d$ holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of $X_n$ is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of $X_s$ is written back with the lowest address that has not been copied from.
  - the value of $X_d$ is written back with the lowest address that has not been copied to.

If the copy is in the backward direction (PSTATE.N == 1), then:
- $X_s$ holds the highest address that the copy is copied from +1.
- $X_d$ holds the highest address that the copy is copied to +1.
- At the end of the instruction:
  - the value of $X_n$ is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of $X_s$ is written back with the highest address that has not been copied from +1.
  - the value of $X_d$ is written back with the highest address that has not been copied to +1.

For CPYETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- $X_n$ is treated as a signed 64-bit number.
- If the copy is in the forward direction ($X_n$ is a negative number), then:
  - $X_n$ holds $-1 \times$ the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$ holds the lowest address that the copy is copied from $-X_n$.
  - $X_d$ holds the lowest address that the copy is made to $-X_n$.
  - At the end of the instruction, the value of $X_n$ is written back with 0.
- If the copy is in the backward direction ($X_n$ is a positive number), then:
  - $X_n$ holds the number of bytes remaining to be copied in the memory copy in total.
  - $X_s$ holds the highest address that the copy is copied from $-X_n+1$.
  - $X_d$ holds the highest address that the copy is copied to $-X_n+1$.
  - At the end of the instruction, the value of $X_n$ is written back with 0.

For CPYETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- $X_n$ holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - $X_s$ holds the lowest address that the copy is copied from.
  - $X_d$ holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of $X_n$ is written back with 0.
    - the value of $X_s$ is written back with the lowest address that has not been copied from.
    - the value of $X_d$ is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - $X_s$ holds the highest address that the copy is copied from +1.
  - $X_d$ holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of $X_n$ is written back with 0.
    - the value of $X_s$ is written back with the highest address that has not been copied from +1.
    - the value of $X_d$ is written back with the highest address that has not been copied to +1.

### Integer

(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 0  | 1  | 0  | op1| 0  | Rs | 0  | 1  | 1  | 1  | 0  | 1  | Rn |  | Rd |

op2
Epilogue (op1 == 10)

CPYETWN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMTWN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPTWN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd>  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs>  For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn>  For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0> + cpysize<55:0>))
        forward = TRUE;
    elseif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
            else
                PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

    if stage != MOPSSStage_Prologue then
      X[n] = cpysize;
  else
    while UInt(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= UInt(stagecpysize);

      if PSTATE.N == '0' then
        readdata<B*8-1:0> = Mem[fromaddress+B, B, racctype];
        Mem[toaddress+B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;

    if stage != MOPSSStage_Prologue then
      X[n] = cpysize;
      X[d] = toaddress;
      X[s] = fromaddress;
if stage == MOPSSstage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
Memory Copy, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWN, the following saturation logic is applied:

If Xn<63:55> != 0000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWN, option A (which results in encoding PSTATE.C = 0):

• PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the forward direction, then:
  ◦ Xs holds the original Xs + saturated Xn.
  ◦ Xd holds the original Xd + saturated Xn.
  ◦ Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• If the copy is in the backward direction, then:
  ◦ Xs and Xd are unchanged.
  ◦ Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWN, option B (which results in encoding PSTATE.C = 1):

• If the copy is in the forward direction, then:
  ◦ Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  ◦ Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is made to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is made to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ■ the value of Xs is written back with the lowest address that has not been copied from.
    ■ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ■ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ■ the value of Xs is written back with the highest address that has not been copied from +1.
    ■ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
  • If the copy is in the forward direction (Xn is a negative number), then:
    ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the lowest address that the copy is copied from -Xn.
    ◦ Xd holds the lowest address that the copy is copied to -Xn.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
  • If the copy is in the backward direction (Xn is a positive number), then:
    ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ Xs holds the highest address that the copy is copied from -Xn+1.
    ◦ Xd holds the highest address that the copy is copied to -Xn+1.
    ◦ At the end of the instruction, the value of Xn is written back with 0.
For CPYEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
  • If the copy is in the forward direction (PSTATE.N == 0), then:
    ◦ Xs holds the lowest address that the copy is copied from.
    ◦ Xd holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the lowest address that has not been copied from.
      ■ the value of Xd is written back with the lowest address that has not been copied to.
  • If the copy is in the backward direction (PSTATE.N == 1), then:
    ◦ Xs holds the highest address that the copy is copied from +1.
    ◦ Xd holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      ■ the value of Xn is written back with 0.
      ■ the value of Xs is written back with the highest address that has not been copied from +1.
      ■ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>
Epilogue (op1 == 10)

CPYEWN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMWN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYWN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>))) then
    forward = TRUE;
  elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)) then
    forward = FALSE;
  else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
  if supports_option_a then
    PSTATE.C = '0';
    PSTATE.N = '0';
    if forward then
      // Copy in the forward direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = Zeros(64) - cpysize;
    else
      PSTATE.C = '1';
      if !forward then
        // Copy in the reverse direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        PSTATE.N = '1';
      else
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);
  else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSStage_Epilogue;
          MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
        else
          if PSTATE.C == '0' then
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSStage_Epilogue;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
stagecysize = cpysize - postsize;

// Check if the parameters to this instruction are valid.
if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = FALSE;
  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
else
  stagecysize = postsize;

// Check if the parameters to the epilogue are valid.
if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
  boolean wrong_option = FALSE;
  boolean from_epilogue = TRUE;
  MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
  while SInt(stagecysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecysize = stagecysize + B;
    else
      assert B <= SInt(stagecysize);
      cpysize = cpysize - B;
      stagecysize = stagecysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
  
if stage != MOPSStage_Prologue then
  X[n] = cpysize;
else
  while UInt(stagecysize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);
    assert B <= UInt(stagecysize);

    if PSTATE.N == '0' then
      readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
      Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress + B;
      toaddress = toaddress + B;
    else
      readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
      Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
      fromaddress = fromaddress - B;
      toaddress = toaddress - B;

    cpysize = cpysize - B;
    stagecysize = stagecysize - B;

if stage != MOPSStage_Prologue then
  X[n] = cpysize;
  X[d] = toaddress;
  X[s] = fromaddress;
if stage == MOPSStage_Prologue then
  X[n] = cpysize;
  X[d] = toaddress;
  X[s] = fromaddress;

Memory Copy, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWT performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWT, the following saturation logic is applied: If \( X_n < 63:55 > ! = 000000000 \), the copy size \( X_n \) is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

- If \( (X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s \), then direction = forward
- Elsif \( (X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d \), then direction = backward
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWT, option A (which results in encoding \( \text{PSTATE.C} = 0 \)):

- \( \text{PSTATE.}\{\text{N,Z,V}\} \) are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated } X_n \).
  - \( X_d \) holds the original \( X_d + \text{saturated } X_n \).
  - \( X_n \) holds \( -1^* \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
- If the copy is in the backward direction, then:
  - \( X_s \) and \( X_d \) are unchanged.
  - \( X_n \) holds the saturated value of \( X_n \) - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWT, option B (which results in encoding \( \text{PSTATE.C} = 1 \)):

- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_n \) holds the saturated \( X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( \text{PSTATE.}\{\text{N,Z,V}\} \) are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_d \) holds the original \( X_d + \text{saturated } X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_n \) holds the saturated \( X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( \text{PSTATE.}\{\text{N,Z,V}\} \) are set to \{1,0,0\}.

For CPYMWT, option A (encoded by \( \text{PSTATE.C} = 0 \)), the format of the arguments is:

- \( X_n \) is treated as a signed 64-bit number.
- If the copy is in the forward direction (\( X_n \) is a negative number), then:
  - \( X_n \) holds \( -1^* \text{the number of bytes remaining to be copied in the memory copy in total} \).
  - \( X_s \) holds the lowest address that the copy is copied from \(-X_n\).
  - \( X_d \) holds the lowest address that the copy is made to \(-X_n\).
  - At the end of the instruction, the value of \( X_n \) is written back with \( -1^* \text{the number of bytes remaining to be copied in the memory copy in total} \).
- If the copy is in the backward direction (\( X_n \) is a positive number), then:
  - \( X_n \) holds the number of bytes remaining to be copied in the memory copy in total.
  - \( X_s \) holds the highest address that the copy is copied from \(-X_n+1\).
  - \( X_d \) holds the highest address that the copy is copied to \(-X_n+1\).
  - At the end of the instruction, the value of \( X_n \) is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWT, option B (encoded by \( \text{PSTATE.C} = 1 \)), the format of the arguments is:

- \( X_n \) holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is made to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with 0.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with 0.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  sz | 0  | 1  | 1  | 1  | 0  | 1  | op1 | 0  | Rs  | 0  | 0  | 0  | 1  | 0  | 1  | Rn  | Rd  |
| op2 |

CPYPWT, CPYMWT, CPYEWT
Epilogue (op1 == 10)

CPYEWT [<Xd>], [<Xs>], [<Xn>]

Main (op1 == 01)

CPYMWT [<Xd>], [<Xs>], [<Xn>]

Prologue (op1 == 00)

CPYPWT [<Xd>], [<Xs>], [<Xn>]

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();

bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(6*N) readdata;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

  boolean forward;
  if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) || (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(toaddress<55:0>) > UInt(fromaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0> + cpysize<55:0>)) then
    forward = FALSE;
  elseif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))) then
    forward = TRUE;
  else
    forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

  if supports_option_a then
    PSTATE.C = '0';
    PSTATE.N = '0';
    if forward then
      // Copy in the forward direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      cpysize = Zeros(64) - cpysize;
    else
      PSTATE.C = '1';
      if !forward then
        // Copy in the reverse direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        PSTATE.N = '1';
      else
        PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';
  // IMP DEF selection of the amount covered by pre-processing.
  stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
  assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();

  if SInt(cpysize) > 0 then
    assert SInt(stagecpysize) <= SInt(cpysize);
  else
    assert SInt(stagecpysize) >= SInt(cpysize);

else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

  // Check if this version is consistent with the state of the call.
  if zero_size_exceptions || SInt(cpysize) != 0 then
    if supports_option_a then
      if PSTATE.C == '1' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
      else
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      if PSTATE.C == '0' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if PSTATE.N == '0' then
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else
          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
          X[d] = toaddress;
          X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
CPYPWTN, CPYMWTN, CPYEWTN

Memory Copy, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and performs an implementation defined amount of the memory copy. CPYMWTN performs an implementation defined amount of the memory copy. CPYEWTN performs the last part of the memory copy.

Note

The inclusion of implementation defined amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTN, the following saturation logic is applied:
If Xn<63:55> != 0000000000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = implementation defined choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation defined.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTN, option A (which results in encoding PSTATE.C = 0):
- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an implementation defined number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an implementation defined number of bytes copied.

After execution of CPYPWTN, option B (which results in encoding PSTATE.C = 1):
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an implementation defined number of bytes copied.
  - Xd holds the original Xd + an implementation defined number of bytes copied.
  - Xn holds the saturated Xn - an implementation defined number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an implementation defined number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an implementation defined number of bytes copied.
  - Xn holds the saturated Xn - an implementation defined number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes to be copied in the memory copy in total.
If the copy is in the forward direction (PSTATE.N == 0), then:
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

If the copy is in the backward direction (PSTATE.N == 1), then:
- Xs holds the highest address that the copy is copied from +1.
- Xd holds the highest address that the copy is copied to +1.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the highest address that has not been copied from +1.
  - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is copied to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes to be copied in the memory copy in total
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

### Integer

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>Rs</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Epilogue ($op1 == 10$)

CPYEWTN $<[Xd]>!, [<Xs>]!, <Xn>!$

Main ($op1 == 01$)

CPYMWTN $<[Xd]>!, [<Xs>]!, <Xn>!$

Prologue ($op1 == 00$)

CPYWWTN $<[Xd]>!, [<Xs>]!, <Xn>!$

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

$<Xd>$ For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

$<Xs>$ For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

$<Xn>$ For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
  if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;

boolean forward;
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) & ( UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &
forward = TRUE;
elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) & (UInt(fromaddress<55:0>) + cpysize<55:0>)
  forward = FALSE;
else
  forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

if supports option a then
  PSTATE.C = '0';
  PSTATE.N = '0';
  if forward then
    // Copy in the forward direction offsets the arguments.
    toaddress = toaddress + cpysize;
    fromaddress = fromaddress + cpysize;
    cpysize = Zeros(64) - cpysize;
  else
    PSTATE.C = '1';
    if !forward then
      // Copy in the reverse direction offsets the arguments.
      toaddress = toaddress + cpysize;
      fromaddress = fromaddress + cpysize;
      PSTATE.N = '1';
    else
      PSTATE.N = '0';
  PSTATE.V = '0';
  PSTATE.Z = '0';

// IMP DEF selection of the amount covered by pre-processing.
stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
if SInt(cpysize) > 0 then
  assert SInt(stagecpysize) <= SInt(cpysize);
else
  assert SInt(stagecpysize) >= SInt(cpysize);
else
  boolean zero_size_exceptions = MemCpyZeroSizeCheck();

// Check if this version is consistent with the state of the call.
if zero_size_exceptions || SInt(cpysize) != 0 then
  if supports option_a then
    if PSTATE.C == '1' then
      boolean wrong_option = TRUE;
      boolean from_epilogue = stage == MOPSStage_Epilogue;
      MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
      if PSTATE.C == '0' then
        boolean wrong_option = TRUE;
        boolean from_epilogue = stage == MOPSStage_Epilogue;
        MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
else
  stagecpysize = postsize;

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);

      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;
    else
      assert B <= SInt(stagecpysize);

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
  else
    while UINT(stagecpysize) > 0 do
      // IMP DEF selection of the block size that is worked on. While many
      // implementations might make this constant, that is not assumed.
      B = CPYSizeChoice(toaddress, fromaddress, cpysize);
      assert B <= UINT(stagecpysize);

      if PSTATE.N == '0' then
        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;
      else
        readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
        Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress - B;
        toaddress = toaddress - B;

      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;

      if stage != MOPSStage_Prologue then
        X[n] = cpysize;
        X[d] = toaddress;
        X[s] = fromaddress;
if stage == MOPSStage_Prologue then
    X[n] = cpysize;
    X[d] = toaddress;
    X[s] = fromaddress;
Memory Copy, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTRN performs the last part of the memory copy.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTRN, the following saturation logic is applied:

If \( X_n < 63:55 \neq 000000000 \), the copy size \( X_n \) is saturated to 0x007FFFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

- If \( (X_s > X_d) \&\& (X_d + \text{saturated } X_n) > X_s \), then direction = forward.
- Elseif \( (X_s < X_d) \&\& (X_s + \text{saturated } X_n) > X_d \), then direction = backward.
- Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{saturated } X_n \).
  - \( X_d \) holds the original \( X_d + \text{saturated } X_n \).
  - \( X_n \) holds \(-1\times \text{saturated } X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied}\).
- If the copy is in the backward direction, then:
  - \( X_s \) and \( X_d \) are unchanged.
  - \( X_n \) holds the saturated value of \( X_n \) - \text{an IMPLEMENTATION DEFINED number of bytes copied}.

After execution of CPYPWTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_n \) holds the saturated \( X_n - \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - PSTATE.\{N,Z,V\} are set to \{0,0,0\}.
- If the copy is in the backward direction, then:
  - \( X_s \) holds the original \( X_s + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_d \) holds the original \( X_d + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - \( X_n \) holds the saturated \( X_n + \text{an IMPLEMENTATION DEFINED number of bytes copied} \).
  - PSTATE.\{N,Z,V\} are set to \{1,0,0\}.

For CPYMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- \( X_n \) is treated as a signed 64-bit number.
- If the copy is in the forward direction (\( X_n \) is a negative number), then:
  - \( X_n \) holds \(-1\times \text{the number of bytes remaining to be copied in the memory copy in total}\).
  - \( X_s \) holds the lowest address that the copy is copied from \(-X_n\).
  - \( X_d \) holds the lowest address that the copy is made to \(-X_n\).
  - At the end of the instruction, the value of \( X_n \) is written back with \(-1\times \text{the number of bytes remaining to be copied in the memory copy in total}\).
- If the copy is in the backward direction (\( X_n \) is a positive number), then:
  - \( X_n \) holds the number of bytes remaining to be copied in the memory copy in total.
  - \( X_s \) holds the highest address that the copy is copied from \(-X_n+1\).
  - \( X_d \) holds the highest address that the copy is copied to \(-X_n+1\).
  - At the end of the instruction, the value of \( X_n \) is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- \( X_n \) holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction ($\text{PSTATE.N} == 0$), then:
  ◦ $Xs$ holds the lowest address that the copy is copied from.
  ◦ $Xd$ holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    - the value of $Xn$ is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of $Xs$ is written back with the lowest address that has not been copied from.
    - the value of $Xd$ is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction ($\text{PSTATE.N} == 1$), then:
  ◦ $Xs$ holds the highest address that the copy is copied from +1.
  ◦ $Xd$ holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    - the value of $Xn$ is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of $Xs$ is written back with the highest address that has not been copied from +1.
    - the value of $Xd$ is written back with the highest address that has not been copied to +1.

For CPYEWTRN, option A (encoded by $\text{PSTATE.C} == 0$), the format of the arguments is:
• $Xn$ is treated as a signed 64-bit number.
  • If the copy is in the forward direction ($\text{PSTATE.N}$ is a negative number), then:
    ◦ $Xn$ holds $-1 \times$ the number of bytes remaining to be copied in the memory copy in total.
    ◦ $Xs$ holds the lowest address that the copy is copied from $-Xn$.
    ◦ $Xd$ holds the lowest address that the copy is made to $-Xn$.
    ◦ At the end of the instruction, the value of $Xn$ is written back with 0.
  • If the copy is in the backward direction ($\text{Xn}$ is a positive number), then:
    ◦ $Xn$ holds the number of bytes remaining to be copied in the memory copy in total.
    ◦ $Xs$ holds the highest address that the copy is copied from $-Xn+1$.
    ◦ $Xd$ holds the highest address that the copy is copied to $-Xn+1$.
    ◦ At the end of the instruction, the value of $Xn$ is written back with 0.

For CPYEWTRN, option B (encoded by $\text{PSTATE.C} == 1$), the format of the arguments is:
• $Xn$ holds the number of bytes to be copied in the memory copy in total.
  • If the copy is in the forward direction ($\text{PSTATE.N} == 0$), then:
    ◦ $Xs$ holds the lowest address that the copy is copied from.
    ◦ $Xd$ holds the lowest address that the copy is copied to.
    ◦ At the end of the instruction:
      - the value of $Xn$ is written back with 0.
      - the value of $Xs$ is written back with the lowest address that has not been copied from.
      - the value of $Xd$ is written back with the lowest address that has not been copied to.
  • If the copy is in the backward direction ($\text{PSTATE.N} == 1$), then:
    ◦ $Xs$ holds the highest address that the copy is copied from +1.
    ◦ $Xd$ holds the highest address that the copy is copied to +1.
    ◦ At the end of the instruction:
      - the value of $Xn$ is written back with 0.
      - the value of $Xs$ is written back with the highest address that has not been copied from +1.
      - the value of $Xd$ is written back with the highest address that has not been copied to +1.

Integer (FEAT_MOPS)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>op1</td>
<td></td>
<td>0</td>
<td>Rs</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{op2}$
Epilogue (op1 == 10)

CPYEWTRN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMWTRN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPWTRN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>) > UInt(toaddress<55:0>))
    forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
            PSTATE.N = '1';
            else
                PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

if supports_option_a then
  while SInt(stagecpysize) != 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = CPYSizeChoice(toaddress, fromaddress, cpysize);

    if SInt(cpysize) < 0 then
      assert B <= -1 * SInt(stagecpysize);
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, racctype] = readdata<B*8-1:0>;
      cpysize = cpysize + B;
      stagecpysize = stagecpysize + B;

    else
      assert B <= SInt(stagecpysize);
      cpysize = cpysize - B;
      stagecpysize = stagecpysize - B;
      readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
      Mem[toaddress+cpysize, B, racctype] = readdata<B*8-1:0>;

    if stage != MOPSStage_Prologue then
      X[n] = cpysize;
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if PSTATE.N == '0' then
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, racctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else
          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, racctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
    end
    X[d] = toaddress;
    X[s] = fromaddress;
if stage == \texttt{MOPSSStage\_Prologue} then
\begin{align*}
X[n] &= \text{cpysize}; \\
X[d] &= \text{toaddress}; \\
X[s] &= \text{fromaddress};
\end{align*}
CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory Copy, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTWN, the following saturation logic is applied:
If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.
After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:
If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward
ElseIf (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward
Else direction = IMPLEMENTATION DEFINED choice between forward and backward.
The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTWN, option A (which results in encoding PSTATE.C = 0):
• PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the forward direction, then:
  ◦ Xs holds the original Xs + saturated Xn.
  ◦ Xd holds the original Xd + saturated Xn.
  ◦ Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
• If the copy is in the backward direction, then:
  ◦ Xs and Xd are unchanged.
  ◦ Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTWN, option B (which results in encoding PSTATE.C = 1):
• If the copy is in the forward direction, then:
  ◦ Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.{N,Z,V} are set to {0,0,0}.
• If the copy is in the backward direction, then:
  ◦ Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  ◦ PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is made to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is made to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.

• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with the number of bytes remaining to be copied in the
      memory copy in total.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• If the copy is in the forward direction (Xn is a negative number), then:
  ◦ Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the lowest address that the copy is copied from -Xn.
  ◦ Xd holds the lowest address that the copy is made to -Xn.
  ◦ At the end of the instruction, the value of Xn is written back with 0.
• If the copy is in the backward direction (Xn is a positive number), then:
  ◦ Xn holds the number of bytes remaining to be copied in the memory copy in total.
  ◦ Xs holds the highest address that the copy is copied from -Xn+1.
  ◦ Xd holds the highest address that the copy is copied to -Xn+1.
  ◦ At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes to be copied in the memory copy in total.
• If the copy is in the forward direction (PSTATE.N == 0), then:
  ◦ Xs holds the lowest address that the copy is copied from.
  ◦ Xd holds the lowest address that the copy is copied to.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with 0.
    ▪ the value of Xs is written back with the lowest address that has not been copied from.
    ▪ the value of Xd is written back with the lowest address that has not been copied to.
• If the copy is in the backward direction (PSTATE.N == 1), then:
  ◦ Xs holds the highest address that the copy is copied from +1.
  ◦ Xd holds the highest address that the copy is copied to +1.
  ◦ At the end of the instruction:
    ▪ the value of Xn is written back with 0.
    ▪ the value of Xs is written back with the highest address that has not been copied from +1.
    ▪ the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz | 0  | 1  | 1  | 1  | 0  | 0  | op1| 0  | Rs | 0  | 1  | 0  | 1  | 0  | 1  | Rn | Rd |
| op2|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Epilogue (op1 == 10)

CPYEWTWN [<Xd>], [<Xs>], <Xn>!

Main (op1 == 01)

CPYMWTWN [<Xd>], [<Xs>], <Xn>!

Prologue (op1 == 00)

CPYPWTWN [<Xd>], [<Xs>], <Xn>!

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set"
;
if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
   For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
   For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
   For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
   For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.
Operation
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d];
bits(64) fromaddress = X[s];
bits(64) cpysize = X[n];
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFF<63:0>;
    boolean forward;
    if (((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)))
        || (UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        < UInt(toaddress<55:0>))) then
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);
    if supports_option_a then
        PSTATE.C = '0';
        PSTATE.N = '0';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            PSTATE.C = '1';
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                PSTATE.N = '1';
            else
                PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros();
    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSStage_Epilogue;
                    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

if stage == MOPSStage_Main then
  stagecpysize = cpysize - postsize;

  // Check if the parameters to this instruction are valid.
  if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = FALSE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else
    stagecpysize = postsize;

  // Check if the parameters to the epilogue are valid.
  if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
    MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
  else

    if supports_option_a then
      while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
          assert B <= -1 * SInt(stagecpysize);
          readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
          Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
          cpysize = cpysize + B;
          stagecpysize = stagecpysize + B;
        else

          assert B <= SInt(stagecpysize);
          cpysize = cpysize - B;
          stagecpysize = stagecpysize - B;
          readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
          Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
        end if
      end while
    else
      while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if PSTATE.N == '0' then
          readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
          Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress + B;
          toaddress = toaddress + B;
        else

          readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
          Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
          fromaddress = fromaddress - B;
          toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
          X[n] = cpysize;
          X[d] = toaddress;
          X[s] = fromaddress;
        end if
      end while
    end if
if stage == MOPSStage_Prologue then
    \( X[n] = \text{cpysize}; \)
    \( X[d] = \text{toaddress}; \)
    \( X[s] = \text{fromaddress}; \)
CRC32, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 1 | 0 | 0 | sz | Rn | Rd

CRC32B (sf == 0 && sz == 00)

CRC32B <Wd>, <Wn>, <Wm>

CRC32H (sf == 0 && sz == 01)

CRC32H <Wd>, <Wn>, <Wm>

CRC32W (sf == 0 && sz == 10)

CRC32W <Wd>, <Wn>, <Wm>

CRC32X (sf == 1 && sz == 11)

CRC32X <Wd>, <Wn>, <Xm>

if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm> Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.
Operation

bits(32) acc = X[n];  // accumulator
bits(size) val = X[m];  // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over \{0,1\} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
**CRC32CB, CRC32CH, CRC32CW, CRC32CX**

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In an Armv8.0 implementation, this is an **OPTIONAL** instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

**Note**

*ID_AA64ISAR0_EL1*.CRC32 indicates whether this instruction is supported.

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>sz</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**CRC32CB (sf == 0 && sz == 00)**

CRC32CB `<Wd>`, `<Wn>`, `<Wm>`

**CRC32CH (sf == 0 && sz == 01)**

CRC32CH `<Wd>`, `<Wn>`, `<Wm>`

**CRC32CW (sf == 0 && sz == 10)**

CRC32CW `<Wd>`, `<Wn>`, `<Wm>`

**CRC32CX (sf == 1 && sz == 11)**

CRC32CX `<Wd>`, `<Wn>`, `<Xm>`

```plaintext
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
- `<Wm>` Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.
Operation

bits(32) acc = \[X[n]\]; \hspace{1em} \text{// accumulator}
bits(size) val = \[X[m]\]; \hspace{1em} \text{// input value}
bits(32) poly = 0x1EDC6F41<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over \{0,1\} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction. No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

Note

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

```
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 1 0 1 0 0 1 1 1 1
```

CRm | op2
---|--
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Operation

`ConsumptionOfSpeculativeDataBarrier();`

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSEL <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSEL <Xd>, <Xn>, <Xm>, <cond>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>  Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>  Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>  Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

\[
\begin{align*}
\text{bits(dataSize) result;} \\
\text{if ConditionHolds(cond) then} \\
\quad \text{result} = X[n]; \\
\text{else} \\
\quad \text{result} = X[m]; \\
\text{X[d] = result;}
\end{align*}
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CINC. This means:

- The encodings in this description are named to match the encodings of CINC.
- The description of CINC gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccc}
sf & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & != & 111 & x & 0 & 1 & 1 & 1 & 1 & 1 & 1 & Rd \\
op & Rm & cond & o2 & Rn \\
\end{array}
\]

32-bit (sf == 0)

CSET <Wd>, <cond>

is equivalent to

CINC <Wd>, WZR, WZR, invert(<cond>)

and is always the preferred disassembly.

64-bit (sf == 1)

CSET <Xd>, <cond>

is equivalent to

CINC <Xd>, XZR, XZR, invert(<cond>)

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | !(=111x) | 0 | 0 | 1 | 1 | 1 | 1 | Rd |
|----|---|---|---|---|---|---|---|---|---|---|---|---|        |   |   |   |   |   |   |    |

32-bit (sf == 0)

CSETM <Wd>, <cond>

is equivalent to

CSINV <Wd>, WZR, WZR, invert(<cond>)

and is always the preferred disassembly.

64-bit (sf == 1)

CSETM <Xd>, <cond>

is equivalent to

CSINV <Xd>, XZR, XZR, invert(<cond>)

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1. This instruction is used by the aliases [CINC](#) and [CSET](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSINC <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINC <Xd>, <Xn>, <Xm>, <cond>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<cond>` is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINC</td>
<td>Rm != '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn != '111111' &amp;&amp; Rn == Rm</td>
</tr>
<tr>
<td>CSET</td>
<td>Rm == '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n];
else
    result = X[m];
    result = result + 1;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV, and CSETM.

### 32-bit (sf == 0)

CSINV <Wd>, <Wn>, <Wm>, <cond>

### 64-bit (sf == 1)

CSINV <Xd>, <Xn>, <Xm>, <cond>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

#### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

#### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINV</td>
<td>Rm != '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn != '11111' &amp;&amp; Rn == Rm</td>
</tr>
<tr>
<td>CSETM</td>
<td>Rm == '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn == '11111'</td>
</tr>
</tbody>
</table>

#### Operation

```
b fists(datasize) result;
if ConditionHolds(cond) then
    result = X[n];
else
    result = X[m];
    result = NOT(result);
X[d] = result;
```

#### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register. This instruction is used by the alias \texttt{CNEG}.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | Rm | cond | 0 | 1 | Rn | Rd |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op | o2 |

32-bit (sf == 0)

\texttt{CSNEG \langle Wd \rangle, \langle Wn \rangle, \langle Wm \rangle, \langle cond \rangle}

64-bit (sf == 1)

\texttt{CSNEG \langle Xd \rangle, \langle Xn \rangle, \langle Xm \rangle, \langle cond \rangle}

integer d = UInt(Rd);
ingenere n = UInt(Rn);
ingenere m = UInt(Rm);
ingenere datasize = if sf == '1' then 64 else 32;

Assembler Symbols

- \texttt{<Wd>}: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<Wn>}: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \texttt{<Wm>}: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \texttt{<Xd>}: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \texttt{<Xn>}: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \texttt{<Xm>}: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \texttt{<cond>}: Is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CNEG}</td>
<td>\texttt{cond} != '111x' &amp;&amp; Rn == Rm</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
if \texttt{ConditionHolds} (cond) then
  result = X[n];
else
  result = X[m];
  result = \texttt{NOT} (result);
  result = result + 1;
X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
Data Cache operation. For more information, see $op0==0b01$, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

```
1 1 0 1 0 1 0 0 0 1 0 1 1 1 0
1 0 1 1 0 0 0 0 0 1 op1 0 1 1 1 CRm op2 Rt
```

DC <dc_op>, <Xt>
is equivalent to
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>

and is the preferred disassembly when SysOp(op1, '0111', CRm, op2) == Sys_DC.

**Assembler Symbols**

<dc_op> is a DC instruction name, as listed for the DC system instruction group, encoded in "op1:CRm:op2":

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;dc_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0110</td>
<td>001</td>
<td>IVAC</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>010</td>
<td>ISW</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>011</td>
<td>IGVAC</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>100</td>
<td>IGSW</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>101</td>
<td>IGDVAC</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>110</td>
<td>IGDSW</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>1010</td>
<td>010</td>
<td>CSW</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>1010</td>
<td>100</td>
<td>CGSW</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>1010</td>
<td>110</td>
<td>CGDSW</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>1110</td>
<td>010</td>
<td>CISW</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>1110</td>
<td>100</td>
<td>CIGSW</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>000</td>
<td>1110</td>
<td>110</td>
<td>CIGDSW</td>
<td>FEAT_MTE2</td>
</tr>
<tr>
<td>011</td>
<td>0100</td>
<td>0001ZVA</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>0100</td>
<td>011</td>
<td>GVA</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>0100</td>
<td>100</td>
<td>GZVA</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1010</td>
<td>001</td>
<td>CVAC</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>1010</td>
<td>011</td>
<td>CGVAC</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1010</td>
<td>101</td>
<td>CGDVAC</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1011</td>
<td>001</td>
<td>CVAU</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>1100</td>
<td>001</td>
<td>CVAP</td>
<td>FEAT_DPB</td>
</tr>
<tr>
<td>011</td>
<td>1100</td>
<td>011</td>
<td>CGVAP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1100</td>
<td>101</td>
<td>CGDVAP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1101</td>
<td>001</td>
<td>CVADP</td>
<td>FEAT_DPB2</td>
</tr>
<tr>
<td>011</td>
<td>1101</td>
<td>011</td>
<td>CGVADP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1101</td>
<td>101</td>
<td>CGDVADP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1110</td>
<td>0001</td>
<td>CIVAC</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>1110</td>
<td>011</td>
<td>CIGVAC</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>1110</td>
<td>101</td>
<td>CIGDVAC</td>
<td>FEAT_MTE</td>
</tr>
</tbody>
</table>

<op1> is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.
Operation

The description of SYS gives the operational pseudocode for this instruction.
DCPS1

Debug Change PE State to EL1, when executed in Debug state:

- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTL R_ELx EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and HCR_EL2.TGE == 1.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see DCPS.

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(LL);

if !Halted() then UNDEFINED;
DCPS2

Debug Change PE State to EL2, when executed in Debug state:
• If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
• Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:
• EL2 if the instruction is executed at an exception level that is not EL3.
• EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:
• ELR_ELx becomes UNKNOWN.
• SPSR_ELx becomes UNKNOWN.
• ESR_ELx becomes UNKNOWN.
• DLR_EL0 and DSPSR_EL0 become UNKNOWN.
• The endianness is set according to SCTLR_ELx EE.

This instruction is UNDEFINED at the following exception levels:
• All exception levels if EL2 is not implemented.
• At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see DCPS.

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(LL);

if !Halted() then UNDEFINED;
DCPS3

Debug Change PE State to EL3, when executed in Debug state:
- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:
- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:
- EDSR.SDD == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see DCPS.

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(LL);

if !Halted() then UNDEFINED;
Data Gathering Hint is a hint instruction that indicates that it is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

```
if !HaveDGHExt() then EndOfInstruction();

Operation

Hint_DGH();
```
Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier*.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
```

**Assembler Symbols**

<option> Specifies the limitation on the barrier operation. Values are:

- **SY**: Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

- **ST**: Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

- **LD**: Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

- **ISH**: Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

- **ISHST**: Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

- **ISHLD**: Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

- **NSH**: Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

- **NSHST**: Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

- **NSHLD**: Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.
OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

<imm>
Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Operation

```
DataMemoryBarrier(domain, types);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DRPS

Debug restore process state

```
1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
```

if !Halted() || PSTATE.EL == EL0 then UNDEFINED;

Operation

```
DRPSInstruction();
```
DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

A DSB instruction with the nXS qualifier is complete when the subset of these memory accesses with the XS attribute set to 0 are complete. It does not require that memory accesses with the XS attribute set to 1 are complete.

This instruction is used by the aliases PSSBB, and SSBB.

It has encodings from 2 classes: Memory barrier and Memory nXS barrier.

Memory barrier

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 1 0 1 0 0 0 1 1 0 1 0 1 1 | CRm | 1 0 0 | 1 1 1 1 1 |

opc

DSB <option> | #<imm>

boolean nXS = FALSE;

DSBAlias alias;
case CRm of
when '0000' alias = DSBAlias_SSBB;
when '0100' alias = DSBAlias_PSSBB;
otherwise alias = DSBAlias_DSB;

MBReqDomain domain;
case CRm<3:2> of
when '00' domain = MBReqDomain_OuterShareable;
when '01' domain = MBReqDomain_Nonshareable;
when '10' domain = MBReqDomain_InnerShareable;
when '11' domain = MBReqDomain_FullSystem;

MBReqTypes types;
case CRm<1:0> of
when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
when '01' types = MBReqTypes_Reads;
when '10' types = MBReqTypes_Writes;
when '11' types = MBReqTypes_All;

Memory nXS barrier

(_FEAT_XS)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 | imm2 | 1 0 0 | 0 1 1 1 1 1 |

DSB <option>nXS| #<imm>

if !HaveFeatXS() then UNDEFINED;
MBReqTypes types = MBReqTypes_All;
boolean nXS = TRUE;
DSBAlias alias = DSBAlias_DSB;
MBReqDomain domain;
case imm2 of
when '00' domain = MBReqDomain_OuterShareable;
when '01' domain = MBReqDomain_Nonshareable;
when '10' domain = MBReqDomain_InnerShareable;
when '11' domain = MBReqDomain_FullSystem;

Assembler Symbols

<option> For the memory barrier variant: specifies the limitation on the barrier operation. Values are:
SY  Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST  Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD  Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH  Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST  Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD  Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH  Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST  Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD  Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH  Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST  Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD  Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

Note

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:

SY  Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm<3:2> = 0b11.
ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b10.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b01.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b00.

<imm>
For the memory barrier variant: is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

For the memory nXS barrier variant: is a 5-bit unsigned immediate, encoded in "imm2":

<table>
<thead>
<tr>
<th>imm2</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>16</td>
</tr>
<tr>
<td>01</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>28</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSSBB</td>
<td>CRm == '0100'</td>
</tr>
<tr>
<td>SSBB</td>
<td>CRm == '0000'</td>
</tr>
</tbody>
</table>

Operation

case alias of
when DSBAlias_SSBB
SpeculativeStoreBypassBarrierToVA();
when DSBAlias_PSSBB
SpeculativeStoreBypassBarrierToPA();
when DSBAlias_DSB
if !nXS && HaveFeatXS() && HaveFeatHCX() then
  nXS = PSTATE.EL IN {EL0, EL1} && IshCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
  DataSynchronizationBarrier(domain, types, nXS);
otherwise
  Unreachable();
DVP

Data Value Prediction Restriction by Context prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see DVP RCTX, Data Value Prediction Restriction by Context.

This is an alias of SYS. This means:

• The encodings in this description are named to match the encodings of SYS.
• The description of SYS gives the operational pseudocode for this instruction.

System
(Feat SpecRES)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 0 1 0 1</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>
```

DVP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #5, <Xt>

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 1 0 1 0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

EON <Xd>, <Xn>, < Xm>{, <shift> #<amount>}

Integer d = UInt(Rd);
Integer n = UInt(Rn);
Integer m = UInt(Rm);
Integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
Integer shift_amount = UInt(imm6);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 EOR operand2;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

32-bit (sf == 0 && N == 0)

EOR <Wd|WSP>, <Wn>, #<imm>

64-bit (sf == 1)

EOR <Xd|SP>, <Xn>, #<imm>

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
result = operand1 EOR imm;
if d == 31 then
   SP[] = result;
else
   X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**EOR (shifted register)**

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 1 0 1 0</th>
<th>shift</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

EOR `<Wd>`, `<Wn>`, `<Wm>{, <shift> #<amount>}`

### 64-bit (sf == 1)

EOR `<Xd>`, `<Xn>`, `< Xm>{, <shift> #<amount>}`

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Operation

```python
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(datasize) result;
result = operand1 EOR operand2;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores `PSTATE` from the SPSR, and branches to the address held in the ELR.
The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

```
1 1 0 1 0 1 1 0 1 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0
```

![Binary representation of ERET](image)

**ERET**

if PSTATE.EL == **EL0** then UNDEFINED;

**Operation**

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];
AArch64.ExceptionReturn(target, SPSR[]);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ERETAA, ERETAB**

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See [Illegal return events from AArch64 state](#).

ERETAA and ERETAB are **UNDEFINED** at EL0.

### Integer (FEAT_PAuth)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A 1 1 0 1 0 1 1 0 1 0 0 1 1 1 1 0 0 0 0 1 M 1 1 1 1 1 1 1 1
```

**ERETAA (M == 0)**

ERETAA

**ERETAB (M == 1)**

ERETAB

if PSTATE.EL == EL0 then UNDEFINED;
boolean use_key_a = (M == '0');
if !HavePACExt() then
  UNDEFINED;

**Operation**

```plaintext
AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target;
if use_key_a then
target = AuthIA(ELR[], SP[], TRUE);
else
target = AuthIB(ELR[], SP[], TRUE);
AArch64.ExceptionReturn(target, SPSR[]);
```
**ESB**

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2. This instruction can be used at all Exception levels and in Debug state. In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a NOP.

**System**

(FEATURE_RAS)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

ESB

```
if !HaveRASExt() then EndOfInstruction();
```

**Operation**

```
SynchronizeErrors();
AArch64_ESB0peration();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64_vESB0peration();
TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**EXTR**

Extract register extracts a register from a pair of registers.
This instruction is used by the alias **ROR (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 0 0 1 1 1</th>
<th>N</th>
<th>0</th>
<th>Rm</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**32-bit (sf == 0 && N == 0 && imms == 0xxxxx)**

EXTR <Wd>, <Wn>, <Wm>, #<lsb>

**64-bit (sf == 1 && N == 1)**

EXTR <Xd>, <Xn>, <Xm>, #<lsb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
integer lsb;
if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = UInt(imms);
```

**Assembler Symbols**

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<lsb>` For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field.
  
  For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROR (immediate)</strong></td>
<td>Rn == Rm</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;
result = concat<lsb+datasize-1:lsb>;
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

**Integer**

* (FEAT_MTE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Xm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Xn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Xd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

**Assembler Symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field.

**Operation**

bits(64) address = if n == 31 then SP[] else X[n];
bits(64) mask = X[m];
bits(4) tag = AArch64.AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d] = mask;
**HINT**

Hint instruction is for the instruction set space that is reserved for architectural hint instructions. Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

```
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0     CRm   op2  1 1 1 1 1
```

**HINT #<imm>**

```systemhintop op;
```

```case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 110'
        if !HaveDGHExt() then EndOfInstruction();  // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
                otherwise EndOfInstruction();
            when '0010 000'
                if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
                op = SystemHintOp_ESB;
            when '0010 001'
                if !HaveStatisticalProfiling() then EndOfInstruction();  // Instruction executes as NOP
                op = SystemHintOp_PSB;
            when '0010 010'
                if !HaveSelfHostedTrace() then EndOfInstruction();  // Instruction executes as NOP
                op = SystemHintOp_TSB;
            when '0010 100'
                op = SystemHintOp_CSDB;
            when '0011 xxx'
                case op2 of
                    when '000' SEE "PACIAZ";
                    when '001' SEE "PACIASP);
                    when '010' SEE "PACIBZ";
                    when '011' SEE "PACIBSP";
                    when '100' SEE "AUTIAZ";
                    when '101' SEE "AUTIASP";
                    when '110' SEE "AUTIBZ";
                    when '111' SEE "AUTIBSP";
                when '0100 xx0'
                    op = SystemHintOp_BTI;
                    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
                    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
                otherwise EndOfInstruction();
```

**Assembler Symbols**

- `<imm>` Is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the “CRm:op2” field. The encodings that are allocated to architectural hint functionality are described in the “Hints” table in the “Index by Encoding”.

---

**HINT**

Page 343
Note

For allocated encodings of "CRm:op2":
- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

Operation

case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
    Hint_WFE(1, WFxType_WFE);
  when SystemHintOp_WFI
    Hint_WFI(1, WFxType_WFI);
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    SynchronizeErrors();
    AArch64.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} & EL2Enabled() then AArch64.vESB0peration();
    TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    SetBTypeNext('00');
  otherwise    // do nothing
HLT

Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

HLT #<imm>

if ESCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
  SetBTypeCompatible(TRUE);

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

Halt(DebugHalt_HaltInstruction);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
HVC

Hypervisor Call causes an exception to EL2. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and SCR_EL3.HCE is set to 0.
- When EL3 is not implemented and HCR_EL2.HCD is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- At EL0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC value 0x16, and the value of the immediate argument.

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

HVC #<imm>

// Empty.

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && (!IsSecureEL2Enabled() && !IsSecure())) then UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then UNDEFINED;
else AArch64.CallHypervisor(imm16);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Instruction Cache operation. For more information, see \textit{op0\textasciitilde 0b01, cache maintenance, TLB maintenance, and address translation instructions}.

This is an alias of \textit{SYS}. This means:

- The encodings in this description are named to match the encodings of \textit{SYS}.
- The description of \textit{SYS} gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>1 1 0 1 0 1 0 0 0 0 1 op1 0 1 1 1 CRm</th>
<th>CRn</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{IC \texttt{&lt;ic_op&gt;}, \texttt{&lt;Xt&gt;}}</td>
<td>\textit{SYS #&lt;op1&gt;, C7, \texttt{&lt;Cm&gt;}, #&lt;op2&gt;\texttt{&lt;Xt&gt;}}</td>
<td>and is the preferred disassembly when \texttt{SysOp(op1,'0111',CRm,op2) \textasciitilde \textit{Sys_IC}}.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

\texttt{<ic\_op>} Is an IC instruction name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>\texttt{&lt;ic_op&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0001</td>
<td>000</td>
<td>IALLUIS</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>000</td>
<td>IALLU</td>
</tr>
<tr>
<td>011</td>
<td>0101</td>
<td>001</td>
<td>IVAU</td>
</tr>
</tbody>
</table>

- \texttt{<op1>} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- \texttt{<Cm>} Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- \texttt{<op2>} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- \texttt{<Xt>} Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

**Operation**

The description of \textit{SYS} gives the operational pseudocode for this instruction.
IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

Integer
(FeAT_MTE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | Xm | 0  | 0  | 0  | 1  | 0  | 0  | Xn | Xd |

IRG <Xd|SP>, <Xn|SP>{, <Xm>}

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

Operation

bits(64) operand = if n == 31 then SP[] else X[n];
bits(64) exclude_reg = X[m];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
  if GCR_EL1.RRND == '1' then
    RGSR_EL1 = bits(64) UNKNOWN;
    if IsOnes(exclude) then
      rtag = '0000';
    else
      rtag = ChooseRandomNonExcludedTag(exclude);
  else
    bits(4) start = RGSR_EL1.TAG;
    bits(4) offset = AArch64.RandomTag();
    rtag = AArch64.ChooseNonExcludedTag(start, offset, exclude);
  RGSR_EL1.TAG = rtag;
else
  rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, AccType_NORMAL, rtag);

if d == 31 then
  SP[] = result;
else
  X[d] = result;
ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).

```
ISB {<option>|<imm>}

// No additional decoding required
```

### Assembler Symbols

- **<option>** Specifies an optional limitation on the barrier operation. Values are:
  - **SY**
    - Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.
    - All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

- **<imm>** Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

### Operation

```
InstructionSynchronizationBarrier();
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, Xt to X(t+7). The data that is loaded is atomic and is required to be 64-byte aligned.

Integer (FEAT_LS64)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | Rn |   |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | Rn |   |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

LD64B <Xt>, [<Xn|SP> {,#0}]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

CheckLDST64Enabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemLoad64B(address, acctype);

for i = 0 to 7
  value = data<63+64*i:64*i>;
  if BigEndian(acctype) then value = BigEndianReverse(value);
  X[t+i] = value;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STADD, STADDL**.

**Integer**

(FEAT_LSE)

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
   1  x 1 1 1 0 0 0 A R 1   Rs 0 0 0 0 0   Rn   Rt
   size  opc
```
32-bit LDADD (size == 10 & A == 0 & R == 0)
LDADD <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDA (size == 10 & A == 1 & R == 0)
LDADDA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDAL (size == 10 & A == 1 & R == 1)
LDADDAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDL (size == 10 & A == 0 & R == 1)
LDADDL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDADD (size == 11 & A == 0 & R == 0)
LDADD <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDA (size == 11 & A == 1 & R == 0)
LDADDA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDAL (size == 11 & A == 1 & R == 1)
LDADDAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL (size == 11 & A == 0 & R == 1)
LDADDL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADD, STADDL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDADDB, LDADDAB, LDADDALB, LDADDLB**

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDADDAB** and **LDADDALB** load from memory with acquire semantics.
- **LDADDB** and **LDADDALB** store to memory with release semantics.
- **LDADDB** has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STADDB, STADDLB**.

#### Integer

<table>
<thead>
<tr>
<th>Size</th>
<th>Opc</th>
<th>Rs</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**LDADDB (A == 1 && R == 0)**

LDADDB <Ws>, <Wt>, [<Xn|SP>]

**LDADDB (A == 0 && R == 0)**

LDADDB <Ws>, <Wt>, [<Xn|SP>]

**LDADDB (A == 0 && R == 1)**

LDADDB <Ws>, <Wt>, [<Xn|SP>]

**LDADDB (A == 1 && R == 1)**

LDADDB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

```
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

#### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDB, STADDLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- LDADDH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.
This instruction is used by the alias STADDH, STADDLH.

### Integer

(Feat_LSE)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>R</td>
</tr>
</tbody>
</table>

**LDADDAH (A == 1 && R == 0)**

LDADDAH <Ws>, <Wt>, [<Xn|SP>]

**LDADDALH (A == 1 && R == 1)**

LDADDALH <Ws>, <Wt>, [<Xn|SP>]

**LDADDH (A == 0 && R == 0)**

LDADDH <Ws>, <Wt>, [<Xn|SP>]

**LDADDLH (A == 0 && R == 1)**

LDADDLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDH, STADDLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(Feat_LRCPC)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | x  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | (1)| (1)| (1)| (1)| 1  | 1  | 0  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rs  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (size == 10)

LDAPR <Wt>, [<Xn|SP> {,#0}]

64-bit (size == 11)

LDAPR <Xt>, [<Xn|SP> {,#0}]

Integer n = UInt(Rn);
Integer t = UInt(Rt);
Integer elsize = 8 << UInt(size);
Integer regsize = if elsize == 64 then 64 else 32;
Boolean tag_checked = n != 31;

Assembler Symbols

- \(<Wt>\) Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.
- \(<Xt>\) Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

\[
\text{bits}(64) \text{ address;}
\text{bits}(\text{elsize}) \text{ data;}
\text{constant integer dbytes = elsize DIV 8;}
\]
\[
\text{if HaveMTE2Ext()} \text{ then}
\text{ SetTagCheckedInstruction(tag checked);}\]
\[
\text{if n == 31 then}
\text{ CheckSPAlignment();}
\text{ address = SP[];}
\text{else}
\text{ address = X[n];}
\]
\[
\text{data = Mem[address, dbytes, AccType.ORDERED];}
\text{X[t] = ZeroExtend(data, regsize);}
\]
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

**Integer (FEAT_LRCPC)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**LDAPRB <Wt>, [<Xn|SP> {,#0}]**

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

**Assembler Symbols**

- *<Wt>* Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- *<Xn|SP>* Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSAPosition();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDAPRH**

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

---

**Integer**  
*(FEAT_LRCPC)*

- **size**
  - `0 1 1 1 0 0 0 1 0 1 (1) (1) (1) (1) 1 1 0 0 0 0`  
- `Rn`  
- `Rt`

**LDAPRH <Wt>, [<Xn|SP> {,#0}]**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```c
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType.ORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a
32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.
The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release,
except that:
• There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release,
  created by having a Store-Release followed by a Load-AcquirePC instruction.
• The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does
  not make the write of the Store-Release globally observed.
This difference in memory ordering is not described in the pseudocode.
For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(DEVICE_LRCPC2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[ \begin{array}{cccccccccccccccccccc}
1 & x & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & Rn & & & Rt \\
\end{array} \]

size opc

32-bit (size == 10)

LDAPUR <Wt>, [<Xn|SP>{, #<simm}>]

64-bit (size == 11)

LDAPUR <Xt>, [<Xn|SP>{, #<simm}>]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in
the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
**Operation**

```plaintext
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(_FEART_LRCPC2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size  
OpC

LDAPURB \(<Wt>, [<Xn|SP>{, #<simm>})

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<simm>\) Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Operation

bits(64) address;
b bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType.ORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode. For information about memory accesses, see *Load/Store addressing modes*.

**Unscaled offset**

(_FEAT_LRCPC2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn | Rn |

`LDAPURH <Wt>, [<Xn|SP>]{, #<simm>}`

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

**Operation**

```java
bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(Feat_LRCPC2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 1  | 1  | x  | 0  | imm9| 0  | 0  | Rn  |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
size| opc|

32-bit (opc == 11)

LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}]

64-bit (opc == 10)

LDAPURSB <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

- <Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```java
int n = UInt(Rn);
int t = UInt(Rt);
MemOp memop;
boolean signed;
int regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH & (n != 31);
```
Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 1, AccType_ORDERED] = data;
  when MemOp_LOAD
    data = Mem[address, 1, AccType_ORDERED];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate
offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.
The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release,
except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release,
  created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does
  not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.
For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(FeaT_LRCPC2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|-------------------|-------------------|
| 0 1 1 0 0 1 1 0 0 1 1 x 0 | imm9              | 0 0               |
| size                      | opc               | Rn                |
| Rt                        |                   |

32-bit (opc == 11)

LDAPURSH <Wt>, [<Xn|SP>{#, #<simm>}]

64-bit (opc == 10)

LDAPURSH <Xt>, [<Xn|SP>{#, #<simm}>]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in
the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_ORDERED];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSW

LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

**Unscaled offset**  
(*FEAT_LRCPC2*)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--]|
LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | (1)(1)(1)(1)(1)(1)(1)(1) | Rn  | Rt  |
| size | L  | Rs | o0 | Rt2 |

32-bit (size == 10)

LDAR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDAR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  |

LDARB <Wt>, [<Xn|SP>{,#0}]

integer n = Uint(Rn);
integer t = Uint(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDARH**

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

**Note**

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 1 0 0 1 0 0 0 1 1 0 (1) (1) (1) (1) | 0 (1) (1) (1) (1) | Rn     | Rt     |
| size          | L    | Rs    | o0    | Rt2   |

LDARH <Wt>, [<Xn|SP>{,0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

**Assembler Symbols**

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```
bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[0];
else
    address = X[n];
data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Annotated for readability.

**Load-Acquire Exclusive Pair of Registers** derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

**32-bit (sz == 0)**

LDAXP `<Wt1>`, `<Wt2>`, `<Xn|SP>{,#0}`

**64-bit (sz == 1)**

LDAXP `<Xt1>`, `<Xt2>`, `<Xn|SP>{,#0}`

```plaintext
text
integer n = UInt(Rn);
text
integer t = UInt(Rt);
text
integer t2 = UInt(Rt2);
text
integer elsize = 32 << UInt(sz);
text
integer datasize = elsize * 2;
text
boolean tag_checked = n != 31;
text
boolean rt_unknown = FALSE;
text
if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
text
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
text
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
text
    when Constraint_NOP EndOfInstruction();
text
```

For information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDAXP*.

**Assembler Symbols**

- `<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;    // In this case t = t2
elseif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType.ORDEREDATOMIC];
    if BigEndian(AccType.ORDEREDATOMIC) then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(AccType.ORDEREDATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType.ORDEREDATOMIC];
    X[t2] = Mem[address+8, 8, AccType.ORDEREDATOMIC];

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

LDAXR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDAXR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

LDAXRB <Wt>, [<Xn|SP>{,1}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDERED_ATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 1 0 0 0 | 0 1 0 | (1) (1) (1) (1) | 1 | (1) (1) (1) (1) |

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>{,,#0} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);

if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDCLR, LDCLA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*
For information about memory accesses see *Load/Store addressing modes.*
This instruction is used by the alias STCLR, STCLRL.

### Integer

(Feat_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 0  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size \(\text{opc}\)
32-bit LDCLR (size == 10 & A == 0 & R == 0)
LDCLR <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRA (size == 10 & A == 1 & R == 0)
LDCLRA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRAL (size == 10 & A == 1 & R == 1)
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRL (size == 10 & A == 0 & R == 1)
LDCLRL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDCLR (size == 11 & A == 0 & R == 0)
LDCLR <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRA (size == 11 & A == 1 & R == 0)
LDCLRA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRAL (size == 11 & A == 1 & R == 1)
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRL (size == 11 & A == 0 & R == 1)
LDCLRL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLR, STCLRL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDCLRAB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRB and LDCLRALB store to memory with release semantics.
- LDCLRB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STCLRAB, STCLRLB.

Integer
(FEAT_LSE)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>Rs</th>
<th>0 0 1 1 0 0 0 0</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 0</td>
<td>1</td>
<td>0 0 0 1 0 0</td>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

LDCLRAB (A == 1 && R == 0)

LDCLRAB <Ws>, <Wt>, [<Xn|SP>]

LDCLRAB (A == 1 && R == 1)

LDCLRAB <Ws>, <Wt>, [<Xn|SP>]

LDCLRB (A == 0 && R == 0)

LDCLRB <Ws>, <Wt>, [<Xn|SP>]

LDCLRB (A == 0 && R == 1)

LDCLRB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLRAB, STCLRLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

This instruction is used by the alias STCLRH, STCLRLH.

### Integer

(\text{FEAT\_LSE})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs  | 0  | 0  | 1  | 0  | Rn  | Rt  |

\text{size} \quad \text{opc}

**LDCLRAH** \((A == 1 \&\& R == 0)\)

\text{LDCLRAH} <W_s>, <W_t>, [<X_n|SP>]

**LDCLRALH** \((A == 1 \&\& R == 1)\)

\text{LDCLRALH} <W_s>, <W_t>, [<X_n|SP>]

**LDCLRH** \((A == 0 \&\& R == 0)\)

\text{LDCLRH} <W_s>, <W_t>, [<X_n|SP>]

**LDCLRLH** \((A == 0 \&\& R == 1)\)

\text{LDCLRLH} <W_s>, <W_t>, [<X_n|SP>]

\text{if !HaveAtomicExt()} \text{then UNDEFINED;}

\text{integer t = UInt(Rt);}
\text{integer n = UInt(Rn);}
\text{integer s = UInt(Rs);}

\text{AccType \_ldacctype = if A == '1' \&\& Rt != '11111' then AccType\_ORDEREDATOMICRW else AccType\__ATOMICRW;}
\text{AccType \_stacctype = if R == '1' \text{then AccType\_ORDEREDATOMICRW else AccType\__ATOMICRW;}\text{boolean tag\_checked = n != 31;}\text{ }

**Assembler Symbols**

- \text{<W_s>} Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \text{<W_t>} Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- \text{<X_n|SP>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLRH, STCLRLH</td>
<td>(A == '0' &amp;&amp; Rt == '11111')</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STEOR, STEORL.

### Integer

**FEAT_LSE**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | Rt |

size | opc

LDEOR, LDEORA, LDEORAL, LDEORL
32-bit LDEOR (size == 10 && A == 0 && R == 0)

LDEOR <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORA (size == 10 && A == 1 && R == 0)

LDEORA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORAL (size == 10 && A == 1 && R == 1)

LDEORAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORL (size == 10 && A == 0 && R == 1)

LDEORL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDEOR (size == 11 && A == 0 && R == 0)

LDEOR <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORA (size == 11 && A == 1 && R == 0)

LDEORA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORAL (size == 11 && A == 1 && R == 1)

LDEORAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORL (size == 11 && A == 0 && R == 1)

LDEORL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR, STEORL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDEOR, LDEORA, LDEORAL, LDEORL
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*

For information about memory accesses see *Load/Store addressing modes.*

This instruction is used by the alias STEORB, STEORLB.

### Integer

**(FEATURE_LSE)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  0 |  0 |  1 |  1 |  0 |  0 | A | R | 1 | Rs | 0 | 0 | 1 | 0 | 0 | 0 | Rn | | | | | | | | | | | | | | | | | | | |
size    opc
```

**LDEORAB** *(A == 1 && R == 0)*

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

**LDEORALB** *(A == 1 && R == 1)*

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

**LDEORB** *(A == 0 && R == 0)*

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

**LDEORLB** *(A == 0 && R == 1)*

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
```

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

```
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

boolean tag_checked = n != 31;

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORB, STEORLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDEORB, LDEORAB, LDEORALB, LDEORLB
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STEORH, STEORLH.

### Integer (FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | 0  | Rn | 1  | Rt |

#### LDEORAH (A == 1 && R == 0)

LDEORAH <Ws>, <Wt>, [<Xn|SP>]

#### LDEORALH (A == 1 && R == 1)

LDEORALH <Ws>, <Wt>, [<Xn|SP>]

#### LDEORH (A == 0 && R == 0)

LDEORH <Ws>, <Wt>, [<Xn|SP>]

#### LDEORLH (A == 0 && R == 1)

LDEORLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORH, STEORLH</td>
<td>A == '0’ &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

Integer
(FEAT_MTE)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>imm9</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>Xn</td>
</tr>
<tr>
<td>22</td>
<td>Xn</td>
</tr>
<tr>
<td>21</td>
<td>Xn</td>
</tr>
<tr>
<td>20</td>
<td>Xn</td>
</tr>
<tr>
<td>19</td>
<td>Xn</td>
</tr>
<tr>
<td>18</td>
<td>Xn</td>
</tr>
<tr>
<td>17</td>
<td>Xn</td>
</tr>
<tr>
<td>16</td>
<td>Xn</td>
</tr>
<tr>
<td>15</td>
<td>Xn</td>
</tr>
<tr>
<td>14</td>
<td>Xn</td>
</tr>
<tr>
<td>13</td>
<td>Xn</td>
</tr>
<tr>
<td>12</td>
<td>Xn</td>
</tr>
<tr>
<td>11</td>
<td>Xn</td>
</tr>
<tr>
<td>10</td>
<td>Xn</td>
</tr>
<tr>
<td>9</td>
<td>X0</td>
</tr>
<tr>
<td>8</td>
<td>X0</td>
</tr>
<tr>
<td>7</td>
<td>X0</td>
</tr>
<tr>
<td>6</td>
<td>X0</td>
</tr>
<tr>
<td>5</td>
<td>X0</td>
</tr>
<tr>
<td>4</td>
<td>X0</td>
</tr>
<tr>
<td>3</td>
<td>X0</td>
</tr>
<tr>
<td>2</td>
<td>X0</td>
</tr>
<tr>
<td>1</td>
<td>X0</td>
</tr>
<tr>
<td>0</td>
<td>X0</td>
</tr>
</tbody>
</table>

LDG <Xt>, [<Xn|SP>{, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

bits(64) address;
bits(4) tag;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
address = Align(address, TAG_GRANULE);
tag = AArch64.MemTag[address, AccType_NORMAL];
X[t] = AArch64.AddressWithAllocationTag(X[t], AccType_NORMAL, tag);
LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4> + 3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is **UNDEFINED** at EL0.

This instruction generates an Unchecked access.

### Integer (FEAT_MTE2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Xn | Xt |

**LDGM <Xt>, [<Xn|SP>]**

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

**Operation**

if PSTATE.EL == EL0 then
UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> \text{LOG2\_TAG\_GRANULE};
integer index = UInt(address<\text{LOG2\_TAG\_GRANULE}+3:\text{LOG2\_TAG\_GRANULE}>);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag(address, AccType\_NORMAL);
data<(index*4)+3:index*4> = tag;
    address = address + \text{TAG\_GRANULE};
    index = index + 1;

X[t] = data;
Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelase. For information about memory accesses, see Load/Store addressing modes.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(FEAT_LOR)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | (1)| (1)| (1)| (1)| (1)| 0  | (1)| (1)| (1)| (1)| (1)|    |    |    |    |    |    |    |    |    |    |
| size | L | Rs | o0 | Rt2 |

32-bit (size == 10)

LDLAR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDLAR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, regsize);
**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDLARB**

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

**Note**

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

**No offset**

**(FEAT_LOR)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |

**Assembler Symbols**

*<Wt>* Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

*<Xn|SP>* Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

```plaintext
LDLARB <Wt>, [<Xn|SP>{,#0}]
```

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDLARH**

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

**Note**

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

**No offset**

*FEAT_LOR*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

Rn  |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |

Rt  |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |

**LDLARH <Wt>, [<Xn|SP>{,#0}]**

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

**Assembler Symbols**

<Wt>   Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see Load/Store addressing modes. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair.

<table>
<thead>
<tr>
<th>opc</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #imm}]

64-bit (opc == 10)

LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #imm}]

// Empty.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    full_data = Mem[address, 2*dbytes, AccType_NORMAL, TRUE];
    if BigEndian(AccType_STREAM) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, AccType_STREAM];
    data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;

X[t] = data1;
X[t2] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 1   | 1   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc |      |      |      |      |      |      |      |      |      | L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

**Pre-index**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | 0   | 1   | 0   | 1   | 0   | 0   | 1   | 1   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc |      |      |      |      |      |      |      |      | L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>]

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>]

boolean wback = TRUE;
boolean postindex = TRUE;

**Signed offset**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | 0   | 1   | 0   | 1   | 0   | 0   | 1   | 0   | 1   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opc |      |      |      |      |      |      |      |      | L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (opc == 00)

LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm}>]

64-bit (opc == 10)

LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

LDP
Operation

bits(64) address;
bids(datasize) data1;
bids(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if HaveLSE2Ext() && !signed then
    bits(2*datasize) full_data;
    full_data = Mem[address, 2*dbytes, AccType_NORMAL, TRUE];
    if BigEndian(AccType_NORMAL) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, AccType_NORMAL];
    data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);
else
    X[t] = data1;
    X[t2] = data2;
if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: **Post-index**,  **Pre-index** and **Signed offset**

**Post-index**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 0 0 1 1 imm7 Rt2 Rn Rt
```

opc L

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;
```

**Pre-index**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 0 1 1 imm7 Rt2 Rn Rt
```

opc L

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!

boolean wback = TRUE;
boolean postindex = FALSE;
```

**Signed offset**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 0 1 0 1 imm7 Rt2 Rn Rt
```

opc L

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONstrained UNPredictable behavior of this instruction, see *Architectural Constraints on UNPredictable behaviors*, and particularly *LDPSW*.

**Assembler Symbols**

< Xt1 > Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

< Xt2 > Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

< Xn|SP > Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

< imm > For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bv64 offset = LSL(SignExtend(imm7, 64), 2);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb bara = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if t == t2 then
  Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
Operation

bits(64) address;
bits(32) data1;
bits(32) data2;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if HaveLSE2Ext() && FALSE then
    bits(64) full_data;
    full_data = Mem[address, 8, AccType_NORMAL, TRUE];
    if BigEndian(AccType_NORMAL) then
        data2 = full_data<31:0>;
        data1 = full_data<63:32>;
    else
        data1 = full_data<31:0>;
        data2 = full_data<63:32>;
    else
        data1 = Mem[address, 4, AccType_NORMAL];
        data2 = Mem[address+4, 4, AccType_NORMAL];

if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
X[t] = SignExtend(data1, 64);
X[t2] = SignExtend(data2, 64);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm9</th>
<th></th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th></th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDR <Wt>, [Xn|SP], #<simm>

64-bit (size == 11)

LDR <Xt>, [Xn|SP], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm9</th>
<th></th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th></th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDR <Wt>, [Xn|SP], #<simm>!

64-bit (size == 11)

LDR <Xt>, [Xn|SP], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm12</th>
<th></th>
<th>Rn</th>
<th></th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDR (immediate)
32-bit (size == 10)

LDR <Wt>, [<Xn|SP>{, #<pimm}]}

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>{, #<pimm}]}

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bite(64) offset = LSL(ZeroExtend(imm12, 64), scale);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDR (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
**Operation**

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | x  | 0 | 1 | 1 | 0 | 0 | 0 | imm19 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

opc

32-bit (opc == 00)

LDR <Wt>, <label>

64-bit (opc == 01)

LDR <Xt>, <label>

integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bite(64) offset;

case opc of
   when '00'
      size = 4;
   when '01'
      size = 8;
   when '10'
      size = 4;
      signed = TRUE;
   when '11'
      memop = MemOp_PREFETCH;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(FALSE);

case memop of
   when MemOp_LOAD
      data = Mem[address, size, AccType_NORMAL];
      if signed then
         X[t] = SignExtend(data, 64);
      else
         X[t] = data;
   when MemOp_PREFETCH
      Prefetch(address, t<4:0>);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes*.

### 32-bit (size == 10)

LDR <Wt>, [<Xn|SP>,, (<Wm>|<Xm>){, <extend> {<amount>}}]

### 64-bit (size == 11)

LDR <Xt>, [<Xn|SP>,, (<Wm>|<Xm>){, <extend> {<amount>}}]

integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;  // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

**Assembler Symbols**

- **<Wt>** Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt>** Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Wm>** When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- **<Xm>** When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- **<extend>** Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- **<amount>** For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB. If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset

(FeaT_PAuth)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | M | S | 1 | imm9 | W | 1 | Rn | Rt |

size

Key A, offset (M == 0 && W == 0)

LDRAA <Xt>, [<Xn|SP>{, #<simm}>]

Key A, pre-indexed (M == 0 && W == 1)

LDRAA <Xt>, [<Xn|SP>{, #<simm}>]!

Key B, offset (M == 1 && W == 0)

LDRAB <Xt>, [<Xn|SP>{, #<simm}>]

Key B, pre-indexed (M == 1 && W == 1)

LDRAB <Xt>, [<Xn|SP>{, #<simm}>]!

if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the “S:imm9” field as <simm>/8.
Operation

bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WB OVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS  wback = FALSE;    // writeback is suppressed
        when Constraint_UNDEF        wb unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_NOP          Undefined()
        when Constraint_UNKNOWN
            wb_unknown = TRUE;    // writeback is UNKNOWN
    if n == 31 then
        address = SP[];
    else
        address = X[n];
    if use_key_a then
        address = AuthDA(address, X[31], TRUE);
    else
        address = AuthDB(address, X[31], TRUE);
    if n == 31 then
        CheckSPAlignment();
    address = address + offset;
    data = Mem[address, 8, AccType_NORMAL];
    X[t] = data;

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 0 1 1 1 0 0 0 0 1 0 | imm9 | 0 1 | Rn | Rt |

Post-index

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 0 1 1 1 0 0 0 0 1 0 | imm9 | 1 1 | Rn | Rt |

Pre-index

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 0 1 1 1 0 0 1 0 1 | imm12 | Rn | Rt |

Unsigned offset

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, ConstraintUNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 1 1 0 0 0 0 1 1 | Rm | option | S | 1 | 0 | Rn | Rt |

Extended register (option != 011)

LDRB <Wt>, [<Xn|SP>, (<Wm>|< Xm>), <extend> {<amount>}]  

Shifted register (option == 011)

LDRB <Wt>, [<Xn|SP>, <Xm> {, LSL <amount>}]  

if option<1> == '0' then UNDEFINED; // sub-word index  
ExtendType extend_type = DecodeRegExtend(option);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.  
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.  
<extend> Is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer m = UInt(Rm);
Operation

```c
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDRH (immediate)**

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: **Post-index**, **Pre-index** and **Unsigned offset**.

### Post-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
|    size         |    opc          |                  |
| imm9           | 0 1             | 0 1              |
| Rn             | Rt              |                  |
```

**LDRH <Wt>, [<Xn|SP>], #<simm>**

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

### Pre-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
|    size         |    opc          |                  |
| imm9           | 1 1             | 0 1              |
| Rn             | Rt              |                  |
```

**LDRH <Wt>, [<Xn|SP>, #<simm>]**

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

### Unsigned offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
|    size         |    opc          |                  |
| imm12          | 0 1 0           | 0 1              |
| Rn             | Rt              |                  |
```

**LDRH <Wt>, [<Xn|SP>{, #<pimm}>]**

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>`: Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as `<pimm>/2`. 

LDRH (immediate)
Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;  // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE;  // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

Operation

```plaintext
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED;  // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;

Assembler Symbols

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>  When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>  When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
b bits(64) address;
b bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | x  | 0  | imm9| 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size  opc

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bias(64) offset = SignExtend(imm9, 64);

Pre-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | x  | 0  | imm9| 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size  opc

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>]

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
bias(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | x  | imm12|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size  opc
32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<pimm>

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<pimm>

boolean wback = FALSE;
boolean postindex = FALSE;
bibliasm(64) offset = LSL(ZeroExtend(imm12, 64), 0);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSB (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
integer n = \texttt{UInt}(Rn);  
integer t = \texttt{UInt}(Rt);  
\texttt{MemOp} memop;  
boolean signed;  
integer regsize;  

if opc<1> == '0' then  
   // store or zero-extending load  
   memop = if opc<0> == '1' then \texttt{MemOp\_LOAD} else \texttt{MemOp\_STORE};  
   regsize = 32;  
   signed = FALSE;  
else  
   // sign-extending load  
   memop = \texttt{MemOp\_LOAD};  
   regsize = if opc<0> == '1' then 32 else 64;  
   signed = TRUE;  

boolean tag_checked = memop != \texttt{MemOp\_PREFETCH} && (wback || n != 31);  

boolean wb_unknown = FALSE;  
boolean rt_unknown = FALSE;  

if memop == \texttt{MemOp\_LOAD} && wback && n == t && n != 31 then  
   c = \texttt{ConstrainUnpredictable(\texttt{Unpredictable\_WBOVERLAPLD})};  
   assert c IN \{\texttt{Constraint\_WBSUPPRESS}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};  
   case c of  
      when \texttt{Constraint\_WBSUPPRESS} wback = FALSE;  // writeback is suppressed  
      when \texttt{Constraint\_UNKNOWN} wb unknown = TRUE;  // writeback is UNKNOWN  
      when \texttt{Constraint\_UNDEF} UNDEFINED;  
      when \texttt{Constraint\_NOP} EndOfInstruction();  

if memop == \texttt{MemOp\_STORE} && wback && n == t && n != 31 then  
   c = \texttt{ConstrainUnpredictable(\texttt{Unpredictable\_WBOVERLAPST})};  
   assert c IN \{\texttt{Constraint\_NONE}, \texttt{Constraint\_UNKNOWN}, \texttt{Constraint\_UNDEF}, \texttt{Constraint\_NOP}\};  
   case c of  
      when \texttt{Constraint\_NONE} rt unknown = FALSE;  // value stored is original value  
      when \texttt{Constraint\_UNKNOWN} rt unknown = TRUE;  // value stored is UNKNOWN  
      when \texttt{Constraint\_UNDEF} UNDEFINED;  
      when \texttt{Constraint\_NOP} EndOfInstruction();  

LDRSB (immediate)
Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 0 1 x 1</td>
<td>Rm</td>
<td>option</td>
</tr>
</tbody>
</table>

32-bit with extended register offset (opc == 11 && option != 011)

LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}]  

32-bit with shifted register offset (opc == 11 && option == 011)

LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

64-bit with extended register offset (opc == 10 && option != 011)

LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}]  

64-bit with shifted register offset (opc == 10 && option == 011)

LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

if option<1> == '0' then UNDEFINED; // sub-word index  

ExtendType extend_type = DecodeRegExtend(option);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.  
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.  
<extend> Is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[);
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
### LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*. It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

#### Post-index

<table>
<thead>
<tr>
<th>32-bit (opc == 11)</th>
<th>64-bit (opc == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRSH &lt;Wt&gt;, [&lt;Xn</td>
<td>SP&gt;], #&lt;simm&gt;</td>
</tr>
</tbody>
</table>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bounds(64) offset = SignExtend(imm9, 64);
```

#### Pre-index

<table>
<thead>
<tr>
<th>32-bit (opc == 11)</th>
<th>64-bit (opc == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRSH &lt;Wt&gt;, [&lt;Xn</td>
<td>SP&gt;], #&lt;simm&gt;!</td>
</tr>
</tbody>
</table>

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bounds(64) offset = SignExtend(imm9, 64);
```

#### Unsigned offset

<table>
<thead>
<tr>
<th>32-bit (opc == 11)</th>
<th>64-bit (opc == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRSH &lt;Wt&gt;, [&lt;Xn</td>
<td>SP&gt;], #&lt;simm&gt;!</td>
</tr>
</tbody>
</table>

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bounds(64) offset = SignExtend(imm12, 64);
```
32-bit (opc == 11)

LDRSH \(<Wt>, [<Xn|SP>{, #<pimm}>]\)

64-bit (opc == 10)

LDRSH \(<Xt>, [<Xn|SP>{, #<pimm}>]\)

boolean wback = FALSE;
boolean postindex = FALSE;
binary(64) offset = LSL(ZeroExtend(imm12, 64), 1);

For information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSH (immediate).

Assembler Symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
\(<Xt>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<\text{simm}>\) Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
\(<\text{pimm}>\) Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as \(<\text{pimm}>/2\).
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb unknown = TRUE;  // writeback is UNKNOWN
    when Constraint_UNDEF  UNDEFINED;
    when Constraint_NOP    EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE   rt unknown = FALSE;  // value stored is original value
    when Constraint_UNKNOWN rt unknown = TRUE;  // value stored is UNKNOWN
    when Constraint_UNDEF   UNDEFINED;
    when Constraint_NOP     EndOfInstruction();
Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then data = bits(16) UNKNOWN;
        else data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then X[t] = SignExtend(data, regsize);
        else X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then address = bits(64) UNKNOWN;
    elsif postindex then address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>Rm</td>
<td>option</td>
</tr>
<tr>
<td>Rn</td>
<td>Rt</td>
</tr>
</tbody>
</table>

32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED;  // sub-word index

ExtendType extend type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;

Assembler Symbols

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt>  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm>  When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm>  When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend>  Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount>  Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 0 0 1 0 0</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

LDRSW <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

### Pre-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 0 0 1 0 0</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

LDRSW <Xt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

### Unsigned offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 0 1 1 0</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

LDRSW <Xt>, [<Xn|SP>{, #<pimm>}

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSW (immediate).

### Assembler Symbols

- <Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- <pimm> Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
Shared Decode

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

Operation

```c
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elseif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDRSW** (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 0  | imm19 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Asm code**

<table>
<thead>
<tr>
<th>opc</th>
<th>LDRSW &lt;Xt&gt;, &lt;label&gt;</th>
</tr>
</thead>
</table>

integer t = UInt(Rt);
bits(64) offset;
offset = SignExtend(imm19:'00', 64);

**Assembler Symbols**

<Xt>  Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<label>  Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19" times 4.

**Operation**

bits(64) address = PC[] + offset;
bits(32) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see Load/Store addressing modes.

![Register Diagram]

LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0;

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSET, STSETL.

Integer

(FEAT_LSE)

```
size  opc
1    x  1  1  1  0  0  0  A  R  1  Rs  0  0  1  1  0  0  Rn  Rt
```

32-bit LDSET (size == 10 & A == 0 & R == 0)
LDSET <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETA (size == 10 & A == 1 & R == 0)
LDSETA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETAL (size == 10 & A == 1 & R == 1)
LDSETAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETL (size == 10 & A == 0 & R == 1)
LDSETL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSET (size == 11 & A == 0 & R == 0)
LDSET <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETA (size == 11 & A == 1 & R == 0)
LDSETA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETAL (size == 11 & A == 1 & R == 1)
LDSETAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETL (size == 11 & A == 0 & R == 1)
LDSETL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSET, STSETL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

```c
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSETB, LDSETAB, LDSETALB, LDSETLB**

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSETB, STSETLB.

---

**Integer (FEAT_LSE)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | Rt |   |   |   |   |   |   |   |   |   |

**LDSETAB (A == 1 && R == 0)**

LDSETAB \(<Ws>, <Wt>, [<Xn|SP]>\)

**LDSETALB (A == 1 && R == 1)**

LDSETALB \(<Ws>, <Wt>, [<Xn|SP]>\)

**LDSETB (A == 0 && R == 0)**

LDSETB \(<Ws>, <Wt>, [<Xn|SP]>\)

**LDSETLB (A == 0 && R == 1)**

LDSETLB \(<Ws>, <Wt>, [<Xn|SP]>\)

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

**AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;**
**AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;**
boolean tag_checked = n != 31;

---

**Assembler Symbols**

\(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

---

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETB, STSETLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);
if t != 31 then
  X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSETH, LDSETAH, LDSETAHL, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETAHL load from memory with acquire semantics.
- LDSETLH and LDSETAHL store to memory with release semantics.
- LDSETH has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.
For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSETH, STSETLH.

**Integer**

(FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs |
| size | opc |

**LDSETH (A == 0 && R == 0)**

LDSETH <Ws>, <Wt>, [<Xn|SP>]

**LDSETLH (A == 0 && R == 1)**

LDSETLH <Ws>, <Wt>, [<Xn|SP>]

**LDSETH (A == 1 && R == 0)**

LDSETH <Ws>, <Wt>, [<Xn|SP>]

**LDSETLH (A == 1 && R == 1)**

LDSETLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt>  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETH, STSETLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL**

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has neither acquire nor release semantics.

For more information about memory ordering semantics see [*Load-Acquire, Store-Release*](#).

For information about memory accesses see [*Load/Store addressing modes*](#).

This instruction is used by the alias [STSMAX, STSMAXL](#).

---

### Integer

**(FEAT_LSE)**

<table>
<thead>
<tr>
<th>Size</th>
<th>Rs</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1  x 1 1 1 0 0 0 A</td>
<td>R</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

This instruction is used by the alias STSMAX, STSMAXL.
32-bit LDSMAX (size == 10 & A == 0 & R == 0)
LDSMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXA (size == 10 & A == 1 & R == 0)
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXAL (size == 10 & A == 1 & R == 1)
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXL (size == 10 & A == 0 & R == 1)
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSMAX (size == 11 & A == 0 & R == 0)
LDSMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXA (size == 11 & A == 1 & R == 0)
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXAL (size == 11 & A == 1 & R == 1)
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXL (size == 11 & A == 0 & R == 1)
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' &amp; Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAX, STSMAXL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXB, STSMAXLB.

### Integer

(FEAT_LSE)

```
<table>
<thead>
<tr>
<th></th>
<th>00110001</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0</td>
<td>010000</td>
</tr>
</tbody>
</table>
```

LDSMAXB (A == 1 && R == 0)

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXAB (A == 1 && R == 1)

LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXB (A == 0 && R == 0)

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXLB (A == 0 && R == 1)

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXB, STSMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Aquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

**Integer**

(\texttt{FEAT\_LSE})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | 0  | 0  | 0  | A  | R  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

- LDSMAXAH (A == 1 && R == 0)
  
  \texttt{LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]}\n
- LDSMAXALH (A == 1 && R == 1)
  
  \texttt{LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]}\n
- LDSMAXH (A == 0 && R == 0)
  
  \texttt{LDSMAXH <Ws>, <Wt>, [<Xn|SP>]}\n
- LDSMAXLH (A == 0 && R == 1)
  
  \texttt{LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]}\n
\texttt{if !HaveAtomicExt() then UNDEFINED;}

integer \( t = \text{UInt}(\text{Rt}); \)

integer \( n = \text{UInt}(\text{Rn}); \)

integer \( s = \text{UInt}(\text{Rs}); \)

\begin{verbatim}
AccType ldacctype = if A == '1' && Rt != '11111' then AccType\_ORDEREDATOMICRW else AccType\__ATOMICRW;
AccType stacctype = if R == '1' then AccType\_ORDEREDATOMICRW else AccType\__ATOMICRW;
boolean tag_checked = n != 31;
\end{verbatim}

**Assembler Symbols**

- \( <\text{Ws}> \) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \( <\text{Wt}> \) Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- \( <\text{Xn}|\text{SP}> \) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXH, STSMAXLH</td>
<td>A == ‘0’ &amp;&amp; Rt == ‘11111’</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
   X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSMIN, LDSMINA, LDSMINAL, LDSMINL**

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMINL and LDSMINAL store to memory with release semantics.
- LDSMIN has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.* For information about memory accesses see *Load/Store addressing modes.* This instruction is used by the alias STSMIN, STSMINL.

**Integer**

(FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |

size opc
32-bit LDSMIN (size == 10 & A == 0 & R == 0)

LDSMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINA (size == 10 & A == 1 & R == 0)

LDSMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINAL (size == 10 & A == 1 & R == 1)

LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINL (size == 10 & A == 0 & R == 1)

LDSMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSMIN (size == 11 & A == 0 & R == 0)

LDSMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINA (size == 11 & A == 1 & R == 0)

LDSMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINAL (size == 11 & A == 1 & R == 1)

LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINL (size == 11 & A == 0 & R == 1)

LDSMINL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMIN, STSMINL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMINB, STSMINLB.

**Integer (FEAT_LSE)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | Rt |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size | opc |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**LDSMINB (A == 1 && R == 0)**

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINALB (A == 1 && R == 1)**

LDSMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINB (A == 0 && R == 0)**

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINLB (A == 0 && R == 1)**

LDSMINLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

```
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

**Assembler Symbols**

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINB, STSMINLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH**

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDSMINAH** and **LDSMINALH** load from memory with acquire semantics.
- **LDSMINLH** and **LDSMINALH** store to memory with release semantics.
- **LDSMINH** has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*

For information about memory accesses see *Load/Store addressing modes.*

This instruction is used by the alias **STSMINH, STSMINLH.**

### Integer

**(FEAT_LSE)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**size**

**opc**

**LDSMINAH (A == 1 && R == 0)**

**LDSMINAH <Ws>, <Wt>, [<Xn|SP>]**

**LDSMINALH (A == 1 && R == 1)**

**LDSMINALH <Ws>, <Wt>, [<Xn|SP>]**

**LDSMINH (A == 0 && R == 0)**

**LDSMINH <Ws>, <Wt>, [<Xn|SP>]**

**LDSMINLH (A == 0 && R == 1)**

**LDSMINLH <Ws>, <Wt>, [<Xn|SP>]**

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

```
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

### Assembler Symbols

**<Ws>**

Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

**<Wt>**

Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

**<Xn|SP>**

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINH, STSMINLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

32-bit (size == 10)

LDTR <Wt>, [<Xn|SP>{, #<simm}>]

64-bit (size == 11)

LDTR <Xt>, [<Xn|SP>{, #<simm}>]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
Operation

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm9</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDTRB** `<Wt>`, `<Xn|SP>{, #<simm}>`  

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
`<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

integer n = UInt(Rn);  
integer t = UInt(Rt);  

```
AccType acctype;  
unpriv_at_el1 = PSTATE.EL == EL1 && !((EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11'));  
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
```

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';  
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then  
  acctype = AccType_UNPRIV;  
else  
  acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

**Operation**

bits(64) address;  
bits(8) data;  

if HaveMTE2Ext() then  
  SetTagCheckedInstruction(tag_checked);

if n == 31 then  
  CheckSPAlignment();  
  address = SP[];
else  
  address = X[n];

address = address + offset;

data = Mem[address, 1, acctype];  
X[t] = ZeroExtend(data, 32);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```plaintext
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

data = Mem[address, 2, acctype];
X[t] = ZeroExtend(data, 32);
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDTRSB**

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1 0 Rn Rt</td>
</tr>
</tbody>
</table>

**32-bit (opc == 11)**

LDTRSB `<Wt>`, [<Xn|SP>{, #<simm>}]

**64-bit (opc == 10)**

LDTRSB `<Xt>`, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer \( n = \) UInt\( \langle Rn \rangle \);
integer \( t = \) UInt\( \langle Rt \rangle \);

```c
AccType acctype;
```

\[
\text{unpriv}_\text{at}_\text{el1} = \text{PSTATE.EL} = \text{EL1} \&\& \neg(\text{EL2Enabled()} \&\& \text{HaveNVExt()} \&\& \text{HCR.EL2.\langle NV, NV1 \rangle} = '1');
\]

\[
\text{unpriv}_\text{at}_\text{el2} = \text{PSTATE.EL} = \text{EL2} \&\& \text{HaveVirtHostExt()} \&\& \text{HCR.EL2.\langle E2H, TG\rangle} = '1';
\]

```
user_access_override = \text{HaveUAOExt()} \&\& \text{PSTATE.UAO} = '1';
if \! user_access_override \&\& \text{unpriv}_\text{at}_\text{el1} || \text{unpriv}_\text{at}_\text{el2} \text{ then}
\hspace{1em} acctype = \text{AccType\_UNPRIV};
else
\hspace{1em} acctype = \text{AccType\_NORMAL};
```

```c
MemOp memop;
boolean signed;
integer regsize;
if \text{opc<1>} = '0' then
\hspace{1em}\hspace{1em}
// store or zero-extending load
\hspace{2em} memop = if \text{opc<0>} = '1' then MemOp\_LOAD else MemOp\_STORE;
\hspace{2em} regsize = 32;
\hspace{2em} signed = FALSE;
else
\hspace{1em}\hspace{1em}
// sign-extending load
\hspace{2em} memop = MemOp\_LOAD;
\hspace{2em} regsize = if \text{opc<0>} = '1' then 32 else 64;
\hspace{2em} signed = TRUE;
```

```
boolean tag_checked = memop \neq \text{MemOp\_PREFETCH} \&\& (n \neq 31);
```

```
Operation
```

bits(64) address;
bv(8) data;
if \text{HaveMTE2Ext()} then
\hspace{1em} SetTagCheckedInstruction\( (\text{tag\_checked}) \);
```

```
if n = 31 then
\hspace{1em} if memop \neq \text{MemOp\_PREFETCH} then CheckSPAlignment();
\hspace{2em} address = SP[{}];
else
\hspace{2em} address = X(n);
```

```
address = address + offset;
```

```
case memop of
\hspace{1em} when \text{MemOp\_STORE}
\hspace{2em} data = X[t];
\hspace{2em} Mem[\text{address, 1, acctype}] = data;
\hspace{1em} when \text{MemOp\_LOAD}
\hspace{2em} data = Mem[\text{address, 1, acctype}];
\hspace{2em} if signed then
\hspace{3em} X[t] = \text{SignExtend}(data, regsize);
\hspace{3em} else
\hspace{4em} X[t] = \text{ZeroExtend}(data, regsize);
```

```
when \text{MemOp\_PREFETCH}
\hspace{1em} Prefetch(address, t<4:0>);
```

```
Operational information
```

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDTRSH**

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**32-bit (opc == 11)**

LDTRSH <Wt>, [<Xn|SP>], #<simm>]

**64-bit (opc == 10)**

LDTRSH <Xt>, [<Xn|SP>], #<simm>]

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- **<Wt>** Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt>** Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<simm>** Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
integer n = UInt(Rn);
integer t = UInt(Rt);

AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '1') &&
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '1';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, acctype] = data;
  when MemOp_LOAD
    data = Mem[address, 2, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | bits(64) offset = SignExtend(imm9, 64); |
| size | opc |

LDTRSW <Xt>, [<Xn|SP>]{, #<simm>}|

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt“ field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn“ field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9“ field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
Accordion acc_type;
unpriv_at_el1 = PSTATE.EL == EL1 && !((EL2Enabled() && HaveNVExt()) && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acc_type = AccType_UNPRIV;
else
  acc_type = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

data = Mem[address, 4, acc_type];
X[t] = SignExtend(data, 64);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- LDUMAX has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAX, STUMAXL.

Integer
(FEAT_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>Rt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size opc
32-bit LDUMAX (size == 10 && A == 0 && R == 0)

LDUMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMAXA (size == 10 && A == 1 && R == 0)

LDUMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMAXAL (size == 10 && A == 1 && R == 1)

LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMAXL (size == 10 && A == 0 && R == 1)

LDUMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDUMAX (size == 11 && A == 0 && R == 0)

LDUMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMAXA (size == 11 && A == 1 && R == 0)

LDUMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMAXAL (size == 11 && A == 1 && R == 1)

LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMAXL (size == 11 && A == 0 && R == 1)

LDUMAXL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '1111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAX, STUMAXL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMALB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.
This instruction is used by the alias STUMAXB, STUMAXLB.

### Integer

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>A</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>Rs</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**LDUMAXB (A == 1 && R == 0)**

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDUMALB (A == 1 && R == 1)**

LDUMALB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXB (A == 0 && R == 0)**

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDUMALB (A == 0 && R == 1)**

LDUMALB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

Accordion ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
Accordion stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

\(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXB, STUMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer (FEAT_LSE)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 0 0 0 A R 1 Rs 0 1 1 0 0 0 Rn Rt |

size opc

LDUMAXAH (A == 1 && R == 0)

LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXALH (A == 1 && R == 1)

LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXH (A == 0 && R == 0)

LDUMAXH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXLH (A == 0 && R == 1)

LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXH, STUMAXLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDUMAXH, LDUMAXAH,  
LDUMAXALH, LDUMAXLH
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMINA and LDUMINAL load from memory with acquire semantics.
- LDUMINL and LDUMINAL store to memory with release semantics.
- LDUMIN has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.
This instruction is used by the alias STUMIN, STUMINL.

Integer
(FEAT_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size
opc
32-bit LDUMIN (size == 10 && A == 0 && R == 0)
LDUMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINA (size == 10 && A == 1 && R == 0)
LDUMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINAL (size == 10 && A == 1 && R == 1)
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINL (size == 10 && A == 0 && R == 1)
LDUMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDUMIN (size == 11 && A == 0 && R == 0)
LDUMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINA (size == 11 && A == 1 && R == 0)
LDUMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINAL (size == 11 && A == 1 && R == 1)
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINL (size == 11 && A == 0 && R == 1)
LDUMINL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == ‘1’ && Rt != ‘11111’ then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == ‘1’ then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMIN, STUMINL</td>
<td>A == ‘0’ &amp; Rt == ‘11111’</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
### LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, **LDUMINAB** and **LDUMINALB** load from memory with acquire semantics.
- **LDUMINLB** and **LDUMINALB** store to memory with release semantics.
- **LDUMINB** has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STUMINB, STUMINLB**.

#### Integer

<table>
<thead>
<tr>
<th>FEAT_LSE</th>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>0 0 1 1 1 0 0 0 A R 1</td>
<td>Rs 0 1 1 1 0 0</td>
</tr>
</tbody>
</table>

**LDUMINAB** (A == 1 && R == 0)

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

**LDUMINALB** (A == 1 && R == 1)

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

**LDUMINB** (A == 0 && R == 0)

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

**LDUMINLB** (A == 0 && R == 1)

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

```
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

#### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINB, STUMINLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bvts(8) value;
bvts(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*
For information about memory accesses see *Load/Store addressing modes.*

This instruction is used by the alias STUMINH, STUMINLH.

**Integer**

(Feat_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

LDUMINAH (A == 1 && R == 0)

LDUMINAH <Ws>, <Wt>, [<Xn|SP>]

LDUMINALH (A == 1 && R == 1)

LDUMINALH <Ws>, <Wt>, [<Xn|SP>]

LDUMINH (A == 0 && R == 0)

LDUMINH <Ws>, <Wt>, [<Xn|SP>]

LDUMINLH (A == 0 && R == 1)

LDUMINLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == ‘1’ && Rt != ‘11111’ then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == ‘1’ then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINH, STUMINLH</td>
<td>A == ‘0’ &amp;&amp; Rt == ‘11111’</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (size == 10)**

LDUR `<Wt>`, `[{Xn|SP}, {, #<simm}>]`

**64-bit (size == 11)**

LDUR `<Xt>`, `[{Xn|SP}, {, #<simm}>]`

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

 `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
 `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
 `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

**Operation**

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rt</th>
<th>opc</th>
<th>imm9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

LDURB <Wt>, [<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = Uint(Rn);
integer t = Uint(Rt);
boolean tag_checked = n != 31;

Operation

bits(64) address;
bites(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
|   0 1 1 1 0 0 0 0 1 0   | imm9            | 0 0             |
| size             | opc             | Rn              |
| Lt               | Rn              | Rt              |
```

LDURH $<Wt>$, $<Xn|SP>$, #<simm>

```plaintext
bits(64) offset = SignExtend(imm9, 64);
```

**Assembler Symbols**

$<Wt>$ Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

$<Xn|SP>$ Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

$<simm>$ Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

**Operation**

```plaintext
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType.NORMAL];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------------------------------------------------------|-----------------|-----------------|
| size | opc |

32-bit (opc == 11)

LDURSB <Wt>, [<Xn|SP>{, #<simm}>]

64-bit (opc == 10)

LDURSB <Xt>, [<Xn|SP>{, #<simm}>]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[);
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 0 0 0 1 x 0 | imm9 | 0 0 | Rn | Rt |
| size | opc |

32-bit (opc == 11)

LDURSH <Wt>, [<Xn|SP>{, #<simm>}

64-bit (opc == 10)

LDURSH <Xt>, [<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;
boolean tag_checked = memop != MemOp_PREFETCH & (n != 31);
Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;
  when MemOp_LOAD
    data = Mem[address, 2, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDURSW**

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

**LDURSW** `<Xt>`, `[<Xn|SP>{, #<simm>}]`

bits(64) offset = `SignExtend`(imm9, 64);

**Assembler Symbols**

- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt“ field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn“ field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9“ field.

**Shared Decode**

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

**Operation**

```java
bits(64) address;
bits(32) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[);
else
    address = X[n];
address = address + offset;
data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit
doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment
requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The PE marks the physical
address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions.
See Synchronization and semaphores. For information about memory accesses, see Load/Store addressing modes.

32-bit (sz == 0)

LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on
UNPREDICTABLE behaviors, and particularly LDXP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
else
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];
// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian(AccType_ATOMIC) then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ATOMIC];

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rs</th>
<th>o0</th>
<th>Rt2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDXR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDXR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 0 | (1) | (1) | (1) | (1) |

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

LDXRH <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
++memory reads from virtual address range [address, address+dbytes-1].
++The Exclusives monitor will only be set if all the reads are from the
++same dbytes-aligned physical address, to allow for the possibility of
++an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LSL (immediate)**

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>!= x11111</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>imms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0 && N == 0 && imms != 011111)

**LSL** `<Wd>, <Wn>, #<shift>`

is equivalent to

**UBFM** `<Wd>, <Wn>, #(<shift> MOD 32), #(31-<shift>)`

and is the preferred disassembly when `imms + 1 == immr`.

### 64-bit (sf == 1 && N == 1 && imms != 111111)

**LSL** `<Xd>, <Xn>, #<shift>`

is equivalent to

**UBFM** `<Xd>, <Xn>, #(<shift> MOD 64), #(63-<shift>)`

and is the preferred disassembly when `imms + 1 == immr`.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<shift>` For the 32-bit variant: is the shift amount, in the range 0 to 31.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.

### Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `< Xm>` is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

### Operation

The description of LSLV gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**LSLV**

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

LSLV <Wd>, <Wn>, <Wm>

**64-bit (sf == 1)**

LSLV <Xd>, <Xn>, <Xm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

**Operational Information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

### 32-bit (sf == 0 & N == 0 & imms == 011111)

```markdown
LSR <Wd>, <Wn>, #<shift>
```

is equivalent to

```markdown
UBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

### 64-bit (sf == 1 & N == 1 & imms == 111111)

```markdown
LSR <Xd>, <Xn>, #<shift>
```

is equivalent to

```markdown
UBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<shift>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

**Operation**

The description of UBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | sf | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |
|------------------------------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

32-bit (sf == 0)

LSR <Wd>, <Wn>, <Wm>

is equivalent to

LSRV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

LSR <Xd>, <Xn>, <Xm>

is equivalent to

LSRV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Right Variable (LSRV) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).

### Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Wd&gt;</td>
<td>Is the 32-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Wn&gt;</td>
<td>Is the 32-bit name of the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt;Wm&gt;</td>
<td>Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the &quot;Rm&quot; field.</td>
</tr>
<tr>
<td>&lt;Xd&gt;</td>
<td>Is the 64-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Xn&gt;</td>
<td>Is the 64-bit name of the first general-purpose source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>&lt; Xm&gt;</td>
<td>Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the &quot;Rm&quot; field.</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);

X[d] = result;
```

### Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 0 0 0</th>
<th>Rm</th>
<th>O</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MADD <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MADD <Xd>, <Xn>, <Xm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa> Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;
result = UInt(operand3) + (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MNEG <Wd>, <Wn>, <Wm>

is equivalent to

MSUB <Wd>, <Wn>, <Wm>, WZR

and is always the preferred disassembly.

64-bit (sf == 1)

MNEG <Xd>, <Xn>, < Xm>

is equivalent to

MSUB <Xd>, <Xn>, < Xm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 0 1 1 0 0 1 0 0 N immr imms 1 1 1 1 1 Rd
```

**operation**

The description of ORR (immediate) gives the operational pseudocode for this instruction.

**Assembler Symbols**

- `<Wd|WSP>` is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<imm>` is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit ($sf == 0 \&\& hw == 0x$)

MOV <Wd>, #<imm>

is equivalent to

MOVN <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when $!(\text{IsZero}(\text{imm16}) \&\& \text{hw} != '00') \&\& !\text{IsOnes}(\text{imm16})$.

64-bit ($sf == 1$)

MOV <Xd>, #<imm>

is equivalent to

MOVN <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when $!(\text{IsZero}(\text{imm16}) \&\& \text{hw} != '00')$.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <imm> For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff.
  For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
- <shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOV (register)**

Move (register) copies the value in a source register to the destination register.

This is an alias of **ORR (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **ORR (shifted register)**.
- The description of **ORR (shifted register)** gives the operational pseudocode for this instruction.

| sf | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rd |
|----|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|---|--|--|--|---|
| opc| shift | N | imm6 | Rn |

32-bit (sf == 0)

MOV <Wd>, <Wm>

is equivalent to

ORR <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

MOV <Xd>, <Xm>

is equivalent to

ORR <Xd>, XZR, <Xm>

and is always the preferred disassembly.

**Assembler Symbols**

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

**Operation**

The description of **ORR (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (to/from SP)

Move between register and stack pointer

: Rd = Rn.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>op</td>
<td>S</td>
<td>sh</td>
<td>imm12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MOV <Wd|WSP>, <Wn|WSP>

is equivalent to

ADD <Wd|WSP>, <Wn|WSP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

64-bit (sf == 1)

MOV <Xd|SP>, <Xn|SP>

is equivalent to

ADD <Xd|SP>, <Xn|SP>, #0

and is the preferred disassembly when (Rd == '11111' || Rn == '11111').

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.
MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

sf | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | hw | imm16 | Rd
opc

32-bit (sf == 0 && hw == 0x)

MOV <Wd>, #<imm>

is equivalent to

MOVZ <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVZ <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm> For the 32-bit variant: is a 32-bit immediate which can be encoded in “imm16:hw”.
For the 64-bit variant: is a 64-bit immediate which can be encoded in “imm16:hw”.

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOVK**

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**opc**

32-bit (sf == 0 && hw == 0x)

```asm
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```asm
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```java
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<imm>` Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- `<shift>` For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as `<shift>/16`.
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as `<shift>/16`.

**Operation**

```java
bits(datasize) result;
result = X[d];
result<pos+15:pos> = imm16;
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOVN**

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register. This instruction is used by the alias **MOV (inverted wide immediate)**.

32-bit (sf == 0 & hw == 0x)

MOVN <Wd>, #<imm>{, LSL #<shift>}

64-bit (sf == 1)

MOVN <Xd>, #<imm>{, LSL #<shift>}

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<imm>** Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- **<shift>** For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
  
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>64-bit</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00')</td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td>! (IsZero(imm16) &amp;&amp; hw != '00') &amp; ! IsOnes(imm16)</td>
</tr>
</tbody>
</table>

**Operation**

```
bits(datasize) result;

result = Zeros();

result<pos+15:pos> = imm16;
result = NOT(result);
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**MOVZ**

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.
This instruction is used by the alias **MOV (wide immediate)**.

| sf | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| hw |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| imm16 |    |
| Rd |

**32-bit (sf == 0 && hw == 0x)**

MOVZ <Wd>, #<imm>{, LSL #<shift>}

**64-bit (sf == 1)**

MOVZ <Xd>, #<imm>{, LSL #<shift>}

```python
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<imm>` Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- `<shift>` For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (wide immediate)</td>
<td>! (IsZero(imm16) &amp; hw != '00')</td>
</tr>
</tbody>
</table>

**Operation**

bits(datasize) result;

result = Zeros();

result<pos+15:pos> = imm16;
X[d] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & o0 & op1 & CRn & CRm & op2 & Rt \\
\end{array}
\]

\( MRS \langle Xt \rangle, (\langle \text{systemreg}\rangle | <\text{op0}> | <\text{op1}> | <\text{Cn}> | <\text{Cm}> | <\text{op2}> ) \)

integer \( t = \text{UInt}(Rt) \);
integer \( \text{sys_op0} = 2 + \text{UInt}(o0) \);
integer \( \text{sys_op1} = \text{UInt}(op1) \);
integer \( \text{sys_op2} = \text{UInt}(op2) \);
integer \( \text{sys_crn} = \text{UInt}(CRn) \);
integer \( \text{sys_crm} = \text{UInt}(CRm) \);

**Assembler Symbols**

\(<Xt>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
\(<\text{systemreg}>\) Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in 'AArch64 System Registers' in the System Register XML.

\(<\text{op0}>\) Is an unsigned immediate, encoded in "o0":

<table>
<thead>
<tr>
<th>(&lt;\text{op0}&gt;)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>
\(\text{sys_op0}\) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
\(\text{Cn}\) Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
\(\text{Cm}\) Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
\(\text{op2}\) Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Operation**

\( \text{AArch64.SysRegRead}(\text{sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t}); \)
MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE*.

The bits that can be written by this instruction are:
- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If *FEAT_SSBS* is implemented, PSTATE.SSBS.
- If *FEAT_PAN* is implemented, PSTATE.PAN.
- If *FEAT_UAO* is implemented, PSTATE.UAO.
- If *FEAT_UAO* is implemented, PSTATE.UAO.
- If *FEAT_DIT* is implemented, PSTATE.DIT.
- If *FEAT_MTE* is implemented, PSTATE.TCO.
- If *FEAT_NMI* is implemented, PSTATE.ALLINT.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>op1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CRm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
  when '00x'
    min_EL = EL1;
  when '010'
    min_EL = EL1;
  when '011'
    min_EL = EL0;
  when '100'
    min_EL = EL2;
  when '101'
    if !HaveVirtHostExt() then
      UNDEFINED;
    min_EL = EL2;
  when '110'
    min_EL = EL3;
  when '111'
    min_EL = EL1;

need_secure = TRUE;

if UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && !IsSecure()) then
  UNDEFINED;

PSTATEField field;
case op1:op2 of
  when '000 011'
    if !HaveUAOExt() then UNDEFINED;
    field = PSTATEField_UAO;
  when '000 100'
    if !HavePANExt() then UNDEFINED;
    field = PSTATEField_PAN;
  when '000 101' field = PSTATEField_SP;
  when '001 000'
    if !HaveFeatNMI() then UNDEFINED;
    if CRm<3:1> != '000' then UNDEFINED;
    field = PSTATEField_ALLINT;
  when '011 010'
    if !HaveDITExt() then UNDEFINED;
    field = PSTATEField_DIT;
  when '011 100'
    if !HaveMTEExt() then UNDEFINED;
    field = PSTATEField_TCO;
  when '011 110' field = PSTATEField_DAIFSet;
  when '011 111' field = PSTATEField_DAIFClr;
  when '011 001'
    if !HaveSSBSExt() then UNDEFINED;
    field = PSTATEField_SSBS;
  otherwise UNDEFINED;

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
  if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);
<pstatefield> Is a PSTATE field name, encoded in “op1:op2:CRm”:

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>CRm</th>
<th>&lt;pstatefield&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00x</td>
<td>xxxx</td>
<td>SEE PSTATE</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>xxxx</td>
<td>SEE PSTATE</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>xxxx</td>
<td>UAO</td>
<td>FEAT_UAO</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>xxxx</td>
<td>PAN</td>
<td>FEAT_PAN</td>
</tr>
<tr>
<td>000</td>
<td>101</td>
<td>xxxx</td>
<td>SPsel</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>11x</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>000x</td>
<td>ALLINT</td>
<td>FEAT_NMI</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>001x</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>01xx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>01x</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>1xx</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>xxx</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>00x</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>xxxx</td>
<td>SSBS</td>
<td>FEAT_SSBS</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>xxxx</td>
<td>DIT</td>
<td>FEAT_DIT</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>100</td>
<td>xxxx</td>
<td>TCO</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>101</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>xxxx</td>
<td>DAIFSet</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>111</td>
<td>xxxx</td>
<td>DAIFClr</td>
<td>-</td>
</tr>
<tr>
<td>1xx</td>
<td>xxx</td>
<td>xxxx</td>
<td>RESERVED</td>
<td>-</td>
</tr>
</tbody>
</table>

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field. Restricted to the range 0 to 1, encoded in "CRm<0>", when <pstatefield> is ALLINT.

**Operation**

case field of
  when PSTATEField_SSBS
    PSTATE.SSBS = CRm<0>;
  when PSTATEField_SP
    PSTATE.SP = CRm<0>;
  when PSTATEField_DAIFSet
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
  when PSTATEField_DAIFClr
    PSTATE.D = PSTATE.D AND NOT(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
  when PSTATEField_PAN
    PSTATE.PAN = CRm<0>;
  when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
  when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
  when PSTATEField_TCO
    PSTATE.TCO = CRm<0>;
  when PSTATEField_ALLINT
    if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> == '1') then
      AArch64.SystemAccessTrap(EL2, 0x18);
    PSTATE.ALLINT = CRm<0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

MSR (<systemreg>|<op0>|<op1>|<Cn>|<Cm>|<op2>), <Xt>

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys.crm = UInt(CRm);

Assembler Symbols

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2". The System register names are defined in 'AArch64 System Registers' in the System Register XML.

<op0> Is an unsigned immediate, encoded in "o0":

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys.crm, sys_op2, t);
MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| sf 0 0 1 1 0 1 0 0 0 | Rm 1 | Ra | Rn | Rd |
| 0 |

32-bit (sf == 0)

MSUB <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MSUB <Xd>, <Xn>, <Xm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa> Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNEG</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL

Multiply

\[ \text{Rd} = \text{Rn} \times \text{Rm} \]

This is an alias of \texttt{MADD}. This means:

- The encodings in this description are named to match the encodings of \texttt{MADD}.
- The description of \texttt{MADD} gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccc}
\hline
\text{sf} & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\text{Rm} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{Rn} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{Rd} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[ 00 \quad \text{Ra} \]

32-bit (sf == 0)

MUL \(<Wd>, \<Wn>, \<Wm>\)

is equivalent to

MADD \(<Wd>, \<Wn>, \<Wm>, \text{WZR}\)

and is always the preferred disassembly.

64-bit (sf == 1)

MUL \(<Xd>, \<Xn>, \< Xm>\)

is equivalent to

MADD \(<Xd>, \<Xn>, \< Xm>, \text{XZR}\)

and is always the preferred disassembly.

**Assembler Symbols**

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of \texttt{MADD} gives the operational pseudocode for this instruction.
MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>shift</th>
<th>imm6</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>N</td>
<td>Rm</td>
<td></td>
<td>imm6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MVN <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

MVN <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- <shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.

### Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>` is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>shift</th>
<th>Rm</th>
<th>imm6</th>
<th>!= 11111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**32-bit (sf == 0)**

NEGS `<Wd>, <Wm>{, <shift> #<amount>}`

is equivalent to

SUBS `<Wd>, WZR, <Wm>{, <shift> #<amount>}`

and is always the preferred disassembly.

**64-bit (sf == 1)**

NEGS `<Xd>, <Xm>{, <shift> #<amount>}`

is equivalent to

SUBS `<Xd>, XZR, <Xm>{, <shift> #<amount>}`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

NGC <Wd>, <Wm>

is equivalent to

SBC <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

NGC <Xd>, <Xm>

is equivalent to

SBC <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

• The encodings in this description are named to match the encodings of SBCS.
• The description of SBCS gives the operational pseudocode for this instruction.

| sf | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rd |
| op | S |

32-bit (sf == 0)

NGCS <Wd>, <Wm>

is equivalent to

SBCS <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

NGCS <Xd>, <Xm>

is equivalent to

SBCS <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 1

NOP

// Empty.

Operation

// do nothing

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias `MVN`.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ORN `<Wd>, <Wn>, <Wm>{,<shift> #<amount>}`

64-bit (sf == 1)

ORN `<Xd>, <Xn>, <Xm>{,<shift> #<amount>}`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 OR operand2;
X[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**ORR (immediate)**

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias **MOV (bitmask immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0 && N == 0)**

ORR <Wd|WSP>, <Wn>, #<imm>

**64-bit (sf == 1)**

ORR <Xd|SP>, <Xn>, #<imm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

**Assembler Symbols**

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>` For the 32-bit variant: is the bitmask immediate, encoded in “imms:immr”.
- For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr”.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (bitmask immediate)</td>
<td>Rn == '1111' &amp;&amp; ! MoveWidePreferred(sf, N, imms, immr)</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
result = operand1 OR imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its registers.
The values of the NZCV flags.
**ORR (shifted register)**

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. This instruction is used by the alias **MOV (register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}  

**64-bit (sf == 1)**

ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (register)</td>
<td>shift == '00' &amp;&amp; imm6 == '000000' &amp;&amp; Rn == '1111'</td>
</tr>
</tbody>
</table>
Operation

\[
\text{bits(datasize)} \text{ operand1} = X[n]; \\
\text{bits(datasize)} \text{ operand2} = \text{ShiftReg}(m, \text{shift_type}, \text{shift_amount}); \\
\text{bits(datasize)} \text{ result}; \\
\text{result} = \text{operand1 OR operand2}; \\
X[d] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A. The address is in the general-purpose register that is specified by <Xd>. The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer (FEAT_PAuth)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PACDA (Z == 0)

PACDA <Xd>, <Xn|SP>

PACDZA (Z == 1 && Rn == 1111)

PACDZA <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
  UNDEFINED;
if Z == '0' then // PACDA
  if n == 31 then source_is_sp = TRUE;
else // PACDZA
  if n != 31 then UNDEFINED;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if source_is_sp then
  X[d] = AddPACDA(X[d], SP[1]);
else
  X[d] = AddPACDA(X[d], X[n]);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

Integer

( FEAT_PAuth )

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

## PACDB (Z == 0)

```c
PACDB <Xd>, <Xn|SP>
```

## PACDZB (Z == 1 && Rn == 11111)

```c
PACDZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
  UNDEFINED;
if Z == '0' then // PACDB
  if n == 31 then source_is_sp = TRUE;
else // PACDZB
  if n != 31 then UNDEFINED;
```

### Assembler Symbols

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

### Operation

```c
if source_is_sp then
  X[d] = AddPACDB(X[d], SP[]);
else
  X[d] = AddPACDB(X[d], X[n]);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 1  | 1  | 0  | Rn | Rd |
```

PACGA <Xd>, <Xn>, <Xm|SP>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UNDEFINED;
if m == 31 then source_is_sp = TRUE;

Assembler Symbols

<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>  Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

if source_is_sp then
    X[d] = AddPACGA(X[n], SP[]);
else
    X[d] = AddPACGA(X[n], X[m]);
**PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA**

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:
- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

It has encodings from 2 classes: Integer and System

### Integer
(\texttt{FEAT\_PAuth})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 0  | 0  | Rn | Rd |

**PACIA (Z == 0)**

PACIA <Xd>, <Xn|SP>

**PACIZA (Z == 1 && Rn == 1111)**

PACIZA <Xd>

boolean source_is_sp = FALSE;
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if !\texttt{HavePACExt}() then
   UNDEFINED;

if Z == '0' then // PACIA
   if n == 31 then source_is_sp = TRUE;
else // PACIZA
   if n != 31 then UNDEFINED;

### System
(\texttt{FEAT\_PAuth})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | x  | 1  | 0  | 0  | x  | 1  | 1  | 1  | 1  | 1  | 1  | CRm | op2 |
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 000'    // PACIAZ
    d = 30;
    n = 31;
  when '0011 001'    // PACIASP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIASP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIXSP());
  when '0001 000'    // PACIA1716
    d = 17;
    n = 16;
  when '0001 010'    SEE "PACIB";
  when '0001 100'    SEE "AUTIA";
  when '0001 110'    SEE "AUTIB";
  when '0011 01x'    SEE "PACIB";
  when '0011 10x'    SEE "AUTIA";
  when '0011 11x'    SEE "AUTIB";
  when '0000 111'    SEE "XPACLRI";
  otherwise SEE "HINT";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

if HavePACExt() then
  if source_is_sp then
    X[d] = AddPACIA(X[d], SP[]);
  else
    X[d] = AddPACIA(X[d], X[n]);
**PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB**

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:
- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: **Integer** and **System**

### Integer

**(FEAT_PAuth)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PACIB (Z == 0)

PACIB <Xd>, <Xn|SP>

### PACIZB (Z == 1 & & Rn == 1111)

PACIZB <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED;
```

### System

**(FEAT_PAuth)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`CRm op2`
PACIB1716 (CRm == 0001 && op2 == 010)

PACIB1716

PACIBSP (CRm == 0011 && op2 == 011)

PACIBSP

PACIBZ (CRm == 0011 && op2 == 010)

PACIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 010'    // PACIBZ
    d = 30;
    n = 31;
  when '0011 011'    // PACIBSP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIEExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIBSP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIXSP());
  when '0001 010'    // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
  if source_is_sp then
    X[d] = AddPACIB(X[d], SP());
  else
    X[d] = AddPACIB(X[d], X[n]);

PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

For information about memory accesses, see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.<type> is one of:

 PLD  Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

 PLI  Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

 PST  Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

 L1  Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

 L2  Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

 L3  Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

 KEEP  Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

 STRM  Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
Operation

bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
endif

if n == 31 then
    address = SP[ ];
else
    address = X[n];
endif

address = address + offset;

Prefetch(address, t<4:0>);

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

integer t = UInt(Rt);
bits(64) offset;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.
	<type> is one of:
	PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
	PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
	PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
	<target> is one of:
	L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
	L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
	L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
	<policy> is one of:
	KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
	STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.
Operation

bits(64) address = \textbf{PC}[\text{]} + \text{offset};

\textbf{if} HaveMTE2Ext() \textbf{then}
\hspace{1em} \textbf{SetTagCheckedInstruction}(FALSE);

\textbf{Prefetch}(address, t<4:0>);
Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

```
|   |   |   |   |   | Rm | option | S | 1 | 0 | Rn | Rt |
|   |   |   |   |   |    |       |  |   |   |    |    |
```

if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;

Assembler Symbols

- `<prfop>` Is the prefetch operation, defined as `<type><target><policy>`. <type> is one of:
  - PLD: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
  - PLI: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
  - PST: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

- `<target>` is one of:
  - L1: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
  - L2: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
  - L3: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

- `<policy>` is one of:
  - KEEP: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
  - STRM: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory. For other encodings of the "Rt" field, use `<imm5>`.

- `<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using `<prfop>`.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

- `<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when `<amount>` is omitted, encoded in "option":

For more information on these prefetch operations, see Prefetch memory.
<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;
Prefetch(address, t<4:0>);
```
Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PRFUM (<prfop>|<imm5>), [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);

Aggsembler Symbols

- `<prfop>` Is the prefetch operation, defined as `<type><target><policy>`. 
- `<type>` is one of:
  - PLD: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
  - PLI: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
  - PST: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
- `<target>` is one of:
  - L1: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
  - L2: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
  - L3: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
- `<policy>` is one of:
  - KEEP: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
  - STRM: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory. For other encodings of the "Rt" field, use `<imm5>`.

- `<imm5>` Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using `<prfop>`.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```plaintext```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

---

**PRFUM**
Operation

bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

### System

*(FEAT_SPE)*

```assembly
if !HaveStatisticalProfiling() then EndOfInstruction();
```

### Operation

```assembly
ProfilingSynchronizationBarrier();
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode for this instruction.

\[
\begin{array}{ccccccccccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

PSSBB is equivalent to DSB #4

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.
Reverse Bits reverses the bit order in a register.

### 32-bit (sf == 0)

\[ \text{RBIT} \langle W_d \rangle, \langle W_n \rangle \]

### 64-bit (sf == 1)

\[ \text{RBIT} \langle X_d \rangle, \langle X_n \rangle \]

integer \( d = \text{UInt}(Rd); \)
integer \( n = \text{UInt}(Rn); \)
integer datasize = if sf == '1' then 64 else 32;

#### Assembler Symbols

- \( <W_d> \) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( <W_n> \) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \( <X_d> \) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( <X_n> \) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

#### Operation

\[
\begin{align*}
\text{bits(datasize) operand} &= X[n]; \\
\text{bits(datasize) result}; \\
\text{for } i = 0 \text{ to } \text{datasize-1} & \text{ result<((datasize-1)\cdot i) = operand<i>; } \\
X[d] &= \text{result}; 
\end{align*}
\]

#### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

```
0 0 0 0 0 0 0 0 | Rn | 0 0 0 0 0 0 0
1 1 0 1 0 1 1 1 | Z  | 1 1 1 1 1 1 0 1
0 1 0 1 1 1 1 1 | op | A  M
```

RET `{<Xn>}`

```
integer n = UInt(Rn);
```

**Assembler Symbols**

`<Xn>` Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

**Operation**

```
bits(64) target = X[n];

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';
BranchTo(target, BranchType_RET, FALSE);
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**RETA, RETAB**

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

**Integer**

**(FEAT_PAuth)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | M  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

**RETA (M == 0)**

RETA

**RETAB (M == 1)**

RETAB

```java
boolean use_key_a = (M == '0');
if !HavePACExt() then
  UNDEFINED;
```

**Operation**

```java
bits(64) target = X[30];
bits(64) modifier = SP[];
if use_key_a then
  target = AuthIA(target, modifier, TRUE);
else
  target = AuthIB(target, modifier, TRUE);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';
BranchTo(target, BranchType_RET, FALSE);
```
REV

Reverse Bytes reverses the byte order in a register. This instruction is used by the pseudo-instruction \texttt{REV64}.

\begin{tabular}{cccccccccccccccccccc}
sf & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x & Rn & Rd \\
\hline
\end{tabular}

**32-bit (sf == 0 \&\& opc == 10)**

\texttt{REV <Wd>, <Wn>}

**64-bit (sf == 1 \&\& opc == 11)**

\texttt{REV <Xd>, <Xn>}

integer \texttt{d = UInt(Rd)};
integer \texttt{n = UInt(Rn)};

integer \texttt{datasize = if sf == '1' then 64 else 32;}

integer \texttt{container\_size;}

case \texttt{opc of}
\begin{itemize}
  \item when '00'
    \texttt{Unreachable();}
  \item when '01'
    \texttt{container\_size = 16;}
  \item when '10'
    \texttt{container\_size = 32;}
  \item when '11'
    \texttt{if sf == '0' then UNDEFINED;}
    \texttt{container\_size = 64;}
\end{itemize}

**Assembler Symbols**

\begin{itemize}
  \item \texttt{<Wd>} is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \texttt{<Wn>} is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
  \item \texttt{<Xd>} is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \texttt{<Xn>} is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
\end{itemize}

**Operation**

bits(datasize) operand = \texttt{X[n]};
bits(datasize) result;

integer \texttt{containers = datasize DIV container\_size;}
integer \texttt{elements\_per\_container = container\_size DIV 8;}
integer \texttt{index = 0;}
integer \texttt{rev\_index;}
for \texttt{c = 0 to containers-1}
  \texttt{rev\_index = index + ((elements\_per\_container - 1) * 8);}
  for \texttt{e = 0 to elements\_per\_container-1}
    result<rev\_index+7:rev\_index> = operand<index+7:index>;
    \texttt{index = index + 8;}
    \texttt{rev\_index = rev\_index - 8;}
\texttt{X[d] = result;}

REV
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**REV16**

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| opc | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

32-bit (sf == 0)

REV16 <Wd>, <Wn>

64-bit (sf == 1)

REV16 <Xd>, <Xn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;

case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
```

**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

```plaintext
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers - 1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container - 1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;

X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |

Rn    Rd
sf    opc

REV32 <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;

Assembler Symbols

Xd    Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Xn    Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
**REV64**

Reverse Bytes reverses the byte order in a 64-bit general-purpose register. When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is optional whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This is a pseudo-instruction of **REV**. This means:

- The encodings in this description are named to match the encodings of **REV**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **REV** gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | Rn |   |   |   |
| sf | opc|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**64-bit**

REV64 <Xd>, <Xn>

is equivalent to

REV <Xd>, <Xn>

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **REV** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer

(FEAT_FlagM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

RMIF <Xn>, <shift>, <mask>

if !HaveFlagManipulateExt() then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift> Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
<mask> Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

bits(4) tmp;
bets(64) tmpreg = X[n];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.

### 32-bit (sf == 0 && N == 0 && imms == 0xxxx)

ROR <Wd>, <Ws>, #<shift>

is equivalent to

EXTR <Wd>, <Ws>, <Ws>, #<shift>

and is the preferred disassembly when Rn == Rm.

### 64-bit (sf == 1 && N == 1)

ROR <Xd>, <Xs>, #<shift>

is equivalent to

EXTR <Xd>, <Xs>, <Xs>, #<shift>

and is the preferred disassembly when Rn == Rm.

**Assembler Symbols**

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Ws> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xs> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <shift> For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.
  
  For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

**Operation**

The description of EXTR gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0)

ROR <Wd>, <Wn>, <Wm>

is equivalent to

RORV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ROR <Xd>, <Xn>, <Xm>

is equivalent to

RORV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias **ROR (register)**.

| sf | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
|-------------------|
| Rm | 0 | 0 | 1 | 0 | 1 | 1 |
| op2 |
| Rn | 32-bit (sf == 0) |
| Rd |

**RORV** 
32-bit (sf == 0)

RORV <Wd>, <Wn>, <Wm>

**64-bit (sf == 1)**

RORV <Xd>, <Xn>, <Xm>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- integer datasize = if sf == '1' then 64 else 32;
- ShiftType shift_type = DecodeShift(op2);

**Assembler Symbols**

- <Wd> is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;

**Operational Information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Speculation Barrier is a barrier that controls speculation. The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

```
if !HaveSBExt() then UNDEFINED;
```

**Operation**

```
SpeculationBarrier();
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SBC**

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias **NGC**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**32-bit (sf == 0)**

SBC <Wd>, <Wn>, <Wm>

**64-bit (sf == 1)**

SBC <Xd>, <Xn>, <Xm>

```plaintext
type integer {
    d = UInt(Rd);
    n = UInt(Rn);
    m = UInt(Rm);
    datasize = if sf == '1' then 64 else 32;
}
```

**Assembler Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGC</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(ddatasize) result;
bits(ddatasize) operand1 = X[n];
bits(ddatasize) operand2 = X[m];
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **NGCS**.

| sf | 1 1 1 1 0 0 0 0 | Rm | 0 0 0 0 0 0 | Rn | Rd |
|    | op S |

**32-bit (sf == 0)**

SBCS `<Wd>, <Wn>, <Wm>`

**64-bit (sf == 1)**

SBCS `<Xd>, <Xn>, <Xm>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGCS</td>
<td>Rn == '1111'</td>
</tr>
</tbody>
</table>

**Operation**

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

**Operational Information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of \( \langle \text{width} \rangle \) bits from the least significant bits of the source register to bit position \( \langle \text{lsb} \rangle \) of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of \text{SBFM}. This means:

- The encodings in this description are named to match the encodings of \text{SBFM}.
- The description of \text{SBFM} gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0)

\text{SBFIZ} \ <Wd>, \ <Wn>, \ #\langle \text{lsb} \rangle, \ #\langle \text{width} \rangle

is equivalent to

\text{SBFM} \ <Wd>, \ <Wn>, \ #(-\langle \text{lsb} \rangle \mod 32), \ #(\langle \text{width} \rangle-1)

and is the preferred disassembly when \( \text{UInt}(\text{imms}) < \text{UInt}(\text{immr}) \).

64-bit (sf == 1 && N == 1)

\text{SBFIZ} \ <Xd>, \ <Xn>, \ #\langle \text{lsb} \rangle, \ #\langle \text{width} \rangle

is equivalent to

\text{SBFM} \ <Xd>, \ <Xn>, \ #(-\langle \text{lsb} \rangle \mod 64), \ #(\langle \text{width} \rangle-1)

and is the preferred disassembly when \( \text{UInt}(\text{imms}) < \text{UInt}(\text{immr}) \).

Assembler Symbols

- \( \langle Wd \rangle \): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle Wn \rangle \): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \( \langle Xd \rangle \): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \( \langle Xn \rangle \): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \( \langle \text{lsb} \rangle \): For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- \( \langle \text{width} \rangle \): For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-\( \langle \text{lsb} \rangle \).
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-\( \langle \text{lsb} \rangle \).

Operation

The description of \text{SBFM} gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBFM

Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.
If \( \text{immr} \) is greater than or equal to \( \text{imms} \), this copies a bitfield of \((\text{imms} - \text{immr}) + 1\) bits starting from bit position \( \text{immr} \) in the source register to the least significant bits of the destination register.
If \( \text{immr} \) is less than \( \text{imms} \), this copies a bitfield of \((\text{imms} + 1)\) bits from the least significant bits of the source register to bit position \((\text{regsize} - \text{immr})\) of the destination register, where \text{regsize} is the destination register size of 32 or 64 bits.
In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit \((sf == 0 \&\& N == 0)\)

SBFM \(<Wd>, <Wn>, \#<immr>, \#<imms>\)

64-bit \((sf == 1 \&\& N == 1)\)

SBFM \(<Xd>, <Xn>, \#<immr>, \#<imms>\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;
if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;
R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

Assembler Symbols

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<immr>\): For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
- \(<imms>\): For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.
  For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR (immediate)</td>
<td>32-bit</td>
<td>imms == '011111'</td>
</tr>
<tr>
<td>ASR (immediate)</td>
<td>64-bit</td>
<td>imms == '111111'</td>
</tr>
</tbody>
</table>
### Operation

```plaintext
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<S>);

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBFX

Signed Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

SBFX <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

SBFM <Wd>, <Wn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

64-bit (sf == 1 && N == 1)

SBFX <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

SBFM <Xd>, <Xn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <lsb> For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- <width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

32-bit (sf == 0)

SDIV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SDIV <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));
X[d] = result<datasize-1:0>;
**SETF8, SETF16**

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value. The PSTATE.C flag is not affected by these instructions.

### Integer

**(FEAT_FlagM)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | sz | 0  | 0  | 1  | 0  | Rn |

\[
sf
\]

**SETF8 (sz == 0)**

SETF8 <Wn>

**SETF16 (sz == 1)**

SETF16 <Wn>

\[
\text{if} \ !\text{HaveFlagManipulateExt}() \text{ then } \text{UNDEFINED};
\]

\[
\text{integer msb} = \text{if} \ \text{sz} == \ '1' \ \text{then} \ 15 \ \text{else} \ 7;
\]

\[
\text{integer n} = \text{UInt}(\text{Rn});
\]

### Assembler Symbols

\(<\text{Wn}>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

### Operation

\[
\text{bits}(32) \ \text{tmpreg} = X[n];
\]

\[
\text{PSTATE.N} = \text{tmpreg}<\text{msb}>;
\]

\[
\text{PSTATE.Z} = \text{if} \ (\text{tmpreg}<\text{msb}:0> == \text{Zeros}(\text{msb} + 1)) \ \text{then} \ '1' \ \text{else} \ '0';
\]

\[
\text{PSTATE.V} = \text{tmpreg}<\text{msb}+1> \ \text{EOR} \ \text{tmpreg}<\text{msb}>;
\]

//PSTATE.C unchanged;

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SETGP, SETGM, SETGE**

Memory Set with tag setting. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGM performs an IMPLEMENTATION DEFINED amount of the memory set. SETGE performs the last part of the memory set.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGP, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGP, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGM, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGM, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGE, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGE, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.
Epilogue (op2 == 1000)

SETGE [<Xd>], <Xn>, <Xs>

Main (op2 == 0100)

SETGM [<Xd>], <Xn>, <Xs>

Prologue (op2 == 0000)

SETGP [<Xd>], <Xn>, <Xs>

if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise UNDEFINED;
if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.
CheckMOPSEnabled();

bits(64) toaddress = $X[d]$;
bits(64) setsize = $X[n]$;
bits(8) data = $X[s]$;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);
if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0xFFFFFFFFFFFFFF0<63:0>;
    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();
    assert stagesetsize<3:0> == '0000';
    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
    else
        bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
        assert postsize<63> == setsize<63> || postsize == Zeros();
        assert postsize<3:0> == '0000';

        boolean zero_size_exceptions = MemSetZeroSizeCheck();
        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(setsize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSSStage_Epilogue;
                        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            if stage == MOPSSStage_Main then
                stagesetsize = setsize - postsize;
                if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                    boolean wrong_option = FALSE;
boolean from_epilogue = FALSE;

MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

else

stagesetsize = postsize;
if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
boolean wrong_option = FALSE;
boolean from_epilogue = TRUE;
MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

if setsize != Align(setsize, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
while SInt(stagesetsize) < 0 do
  // IMP DEF selection of the block size that is worked on. While many
  // implementations might make this constant, that is not assumed.
  B = SETSizeChoice(toaddress, setsize, 16);
  assert B <= -1 * SInt(stagesetsize);
  assert B<3:0> == '0000';
  Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

  tagstep = B DIV 16;
  tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
  while tagstep > 0 do
    tagaddr = toaddress + setsize + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, acctype] = tag;
    tagstep = tagstep - 1;

  setsize = setsize + B;
  stagesetsize = stagesetsize + B;
if stage != MOPSStage_Prologue then
  X[n] = setsize;
else
while UInt(stagesetsize) > 0 do
  // IMP DEF selection of the block size that is worked on. While many
  // implementations might make this constant, that is not assumed.
  B = SETSizeChoice(toaddress, setsize, 16);
  assert B <= UInt(stagesetsize);
  assert B<3:0> == '0000';
  Mem[toaddress, B, acctype] = Replicate(data, B);

  tagstep = B DIV 16;
  tag = AArch64.AllocationTagFromAddress(toaddress);
  while tagstep > 0 do
    tagaddr = toaddress + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, acctype] = tag;
    tagstep = tagstep - 1;

  toaddress = toaddress + B;
  setsize = setsize - B;
  stagesetsize = stagesetsize - B;
if stage != MOPSStage_Prologue then
  X[n] = setsize;
  X[d] = toaddress;
if stage == MOPSStage_Prologue then
\( X[n] = \text{setsize}; \)
\( X[d] = \text{toaddress}; \)
**SETGPN, SETGMN, SETGEN**

Memory Set with tag setting, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGEN performs the last part of the memory set.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

**Integer**

(Feat_Mops)
Epilogue (op2 == 1010)

SETGEN [<Xd>], <Xn>, <Xs>

Main (op2 == 0110)

SETGMN [<Xd>], <Xn>, <Xs>

Prologue (op2 == 0010)

SETGPN [<Xd>], <Xn>, <Xs>

if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSSStage stage;
case op2<3:2> of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.
CheckMOPSEnabled();

bits(64) toaddress = X[d];
bits(64) setsize = X[n];
bits(8) data = X[s];
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);
boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);
if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;
    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();
    assert stagesetsize<3:0> == '0000';
    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros();
    assert postsize<3:0> == '0000';
    boolean zero_size_exceptions = MemSetZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        if stage == MOPSSStage_Main then
            stagesetsize = setsize - postsize;
            if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                boolean wrong_option = FALSE;
boolean from_epilogue = FALSE;
MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
else
stagesetsize = postsize;
if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
    boolean wrong_option = FALSE;
    boolean from_epilogue = TRUE;
   MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
    boolean iswrite = TRUE;
    boolean secondstage = FALSE;
    AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

if setsize != Align(setsize, TAG_GRANULE) then
    boolean iswrite = TRUE;
    boolean secondstage = FALSE;
    AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n] = setsize;

else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n] = setsize;
            X[d] = toaddress;

if stage == MOPSStage_Prologue then
\[ X[n] = \text{setsize}; \]
\[ X[d] = \text{toaddress}; \]
SETGPT, SETGMT, SETGET

Memory Set with tag setting, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETGET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPT, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.(N,Z,V) are set to {0,0,0}.

After execution of SETGPT, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.(N,Z,V) are set to {0,0,0}.

For SETGMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGET, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGET, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)
Epilogue (op2 == 1001)

```
SETGET [<Xd>], <Xn>, <Xs>
```

Main (op2 == 0101)

```
SETGMT [<Xd>], <Xn>, <Xs>
```

Prologue (op2 == 0001)

```
SETGPT [<Xd>], <Xn>, <Xs>
```

```java
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSSStage stage;
case op2<3:2> of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

- `<Xd>` For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

- `<Xn>` For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

- `<Xs>` For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

- `<Xn>` For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

- `<Xs>` For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.
CheckMOPSEnabled();

bits(64) toaddress = X[d];
bits(64) setsize = X[n];
bits(8) data = X[s];
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSSstage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();
    assert stagesetsize<3:0> == '0000';

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);

    else
        bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
        assert postsize<63> == setsize<63> || postsize == Zeros();
        assert postsize<3:0> == '0000';

        boolean zero_size_exceptions = MemSetZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(setsize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSSStage_Epilogue;
                        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

            if stage == MOPSSstage_Main then
                stagesetsize = setsize - postsize;
                if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                    boolean wrong_option = FALSE;
boolean from_epilogue = FALSE;
MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
else
stagesetsize = postsize;
if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
boolean wrong_option = FALSE;
boolean from_epilogue = TRUE;
MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
if setsize != Align(setsize, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = TRUE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';
        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;
        end
    end
    toaddress = toaddress + B;
    setsize = setsize - B;
    stagesetsize = stagesetsize - B;
    if stage != MOPSStage_Prologue then
        X[n] = setsize;
        X[d] = toaddress;
    end
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';
        Mem[toaddress, B, acctype] = Replicate(data, B);
        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;
        end
        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n] = setsize;
            X[d] = toaddress;
        end
end
\( X[n] = \text{setsize}; \)

\( X[d] = \text{toaddress}; \)
SETGPTN, SETGMTN, SETGETN

Memory Set with tag setting, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGETN performs the last part of the memory set.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPTN, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPTN, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.
Epilogue (op2 == 1011)

SETGETN [<Xd>], <Xn>, <Xs>

Main (op2 == 0111)

SETGMTN [<Xd>], <Xn>, <Xs>

Prologue (op2 == 0011)

SETGPTN [<Xd>], <Xn>, <Xs>

```plaintext
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);

bits(2) options = op2<1:0>;

MOPSStage stage;

case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

**<Xd>**
For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

**<Xn>**
For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

**<Xs>**
For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.
Operation
CheckMOPSEnabled();

bits(64) toaddress = X[d];
bits(64) setsize = X[n];
bits(8) data = X[s];
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCopyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFF0<63:0>;

        if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
            boolean iswrite = TRUE;
            boolean secondstage = FALSE;
            AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

        if setsize != Align(setsize, TAG_GRANULE) then
            boolean iswrite = TRUE;
            boolean secondstage = FALSE;
            AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

        stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
        assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();
        assert stagesetsize<3:0> == '0000';

        if SInt(setsize) > 0 then
            assert SInt(stagesetsize) <= SInt(setsize);
        else
            assert SInt(stagesetsize) >= SInt(setsize);

    else
        bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
        assert postsize<63> == setsize<63> || postsize == Zeros();
        assert postsize<3:0> == '0000';

        boolean zero_size_exceptions = MemSetZeroSizeCheck();

        // Check if this version is consistent with the state of the call.
        if zero_size_exceptions || SInt(setsize) != 0 then
            if supports_option_a then
                if PSTATE.C == '1' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
                else
                    if PSTATE.C == '0' then
                        boolean wrong_option = TRUE;
                        boolean from_epilogue = stage == MOPSSStage_Epilogue;
                        MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

            if stage == MOPSSStage_Main then
                stagesetsize = setsize - postsize;
            if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                boolean wrong_option = FALSE;
boolean from_epilogue = FALSE;
MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
else
stagesetsize = postsize;
if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
boolean wrong_option = FALSE;
boolean from_epilogue = TRUE;
MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));
if setsize != Align(setsize, TAG_GRANULE) then
boolean iswrite = TRUE;
boolean secondstage = FALSE;
AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;
if supports_option_a then
while SInt(stagesetsize) < 0 do
  // IMP DEF selection of the block size that is worked on. While many
  // implementations might make this constant, that is not assumed.
  B = SETSizeChoice(toaddress, setsize, 16);
  assert B <= -1 * SInt(stagesetsize);
  assert B<3:0> == '0000';
  Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
  tagstep = B DIV 16;
  tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
  while tagstep > 0 do
    tagaddr = toaddress + setsize + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, acctype] = tag;
    tagstep = tagstep - 1;
  setsize = setsize + B;
  stagesetsize = stagesetsize + B;
  if stage != MOPSStage_Prologue then
    X[n] = setsize;
else
while UInt(stagesetsize) > 0 do
  // IMP DEF selection of the block size that is worked on. While many
  // implementations might make this constant, that is not assumed.
  B = SETSizeChoice(toaddress, setsize, 16);
  assert B <= UInt(stagesetsize);
  assert B<3:0> == '0000';
  Mem[toaddress, B, acctype] = Replicate(data, B);
  tagstep = B DIV 16;
  tag = AArch64.AllocationTagFromAddress(toaddress);
  while tagstep > 0 do
    tagaddr = toaddress + (tagstep - 1) * 16;
    AArch64.MemTag[tagaddr, acctype] = tag;
    tagstep = tagstep - 1;
  toaddress = toaddress + B;
  setsize = setsize - B;
  stagesetsize = stagesetsize - B;
  if stage != MOPSStage_Prologue then
    X[n] = setsize;
    X[d] = toaddress;
if stage == MOPSStage_Prologue then
\( \mathbf{X}[n] = \text{setsize} \);
\( \mathbf{X}[d] = \text{toaddress} \);
**SETP, SETM, SETE**

Memory Set. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETM performs an IMPLEMENTATION DEFINED amount of the memory set. SETE performs the last part of the memory set.

**Note**

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

**Note**

Portable software should not assume that the choice of algorithm is constant.

After execution of SETP, option A (which results in encoding PSTATE.C = 0):
- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETP, option B (which results in encoding PSTATE.C = 1):
- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETM, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETM, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

For SETE, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETE, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

**Integer (FEAT_MOPS)**

| sz | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | Rs | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
|----|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|----|----|
| op2 |
Epilogue (op2 == 1000)

SETE [<Xd>], [<Xn>], [<Xs>]

Main (op2 == 0100)

SETM [<Xd>], [<Xn>], [<Xs>]

Prologue (op2 == 0000)

SETP [<Xd>], [<Xn>], [<Xs>]

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSSStage stage;
case op2<3:2> of
    when '00' stage = MOPSSStage_Prologue;
    when '01' stage = MOPSSStage_Main;
    when '10' stage = MOPSSStage_Epilogue;
    otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler Symbols

Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
CheckMOPSEnabled();

bits(64) toaddress = X[d];
b无可得(64) setsize = X[n];
b无可得(8) data = X[s];
b无可得(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x007FFFFFFFFFFFF0<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros();

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

        else
            if stage == MOPSSStage_Main then
                stagesetsize = setsize - postsize;
            if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                stagesetsize = postsize;
            if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = TRUE;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

        else
            if supports_option_a then
                while SInt(stagesetsize) < 0 do
                    // IMP DEF selection of the block size that is worked on. While many


// implementations might make this constant, that is not assumed.

B = SETSizeChoice(toaddress, setsize, 1);

assert B <= -1 * SInt(stagesetsize);

Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

setsize = setsize + B;
stagesetsize = stagesetsize + B;

if stage != MOPSStage_Prologue then
    X[n] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, acctype] = Replicate(data, B);
        toaddress = toaddress + B;
        setsize = setsize - B;
stagesetsize = stagesetsize - B;

        if stage != MOPSStage_Prologue then
            X[n] = setsize;
            X[d] = toaddress;
        end
    end

if stage == MOPSStage_Prologue then
    X[n] = setsize;
    X[d] = toaddress;
SETPN, SETMN, SETEN

Memory Set, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPN, option A (which results in encoding PSTATE.C = 0):
- If \( X_n<63> == 1 \), the set size is saturated to \( 0x7FFFFFFFFFFFFFFF \).
- \( X_d \) holds the original \( X_d \) + saturated \( X_n \).
- \( X_n \) holds \(-1^*\) saturated \( X_n \) + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

After execution of SETPN, option B (which results in encoding PSTATE.C = 1):
- If \( X_n<63> == 1 \), the copy size is saturated to \( 0x7FFFFFFFFFFFFFFF \).
- \( X_d \) holds the original \( X_d \) + an IMPLEMENTATION DEFINED number of bytes set.
- \( X_n \) holds the saturated \( X_n \) - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

For SETMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- \( X_n \) is treated as a signed 64-bit number.
- \( X_n \) holds \(-1^*\) number of bytes remaining to be set in the memory set in total.
- \( X_d \) holds the lowest address that the set is made to - \( X_n \).
- At the end of the instruction, the value of \( X_n \) is written back with \(-1^*\) the number of bytes remaining to be set in the memory set in total.

For SETMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- \( X_n \) holds the number of bytes remaining to be set in the memory set in total.
- \( X_d \) holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of \( X_n \) is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of \( X_d \) is written back with the lowest address that has not been set.

For SETEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
- \( X_n \) is treated as a signed 64-bit number.
- \( X_n \) holds \(-1^*\) number of bytes remaining to be set in the memory set in total.
- \( X_d \) holds the lowest address that the set is made to - \( X_n \).
- At the end of the instruction, the value of \( X_n \) is written back with \(0\).

For SETEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
- \( X_n \) holds the number of bytes remaining to be set in the memory set in total.
- \( X_d \) holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of \( X_n \) is written back with \(0\).
  - the value of \( X_d \) is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sz| 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rs| x  | x  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rd|   |   |   |   |   |   | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

op2
Epilogue (op2 == 1010)

SETEN [Xd], <Xn>, <Xs>

Main (op2 == 0110)

SETMN [Xd], <Xn>, <Xs>

Prologue (op2 == 0010)

SETPN [Xd], <Xn>, <Xs>

if !HaveFeatMOPS() then UNDEFINED;
if sz !='00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
  otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd>     For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.
          For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn>     For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.
          For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
          For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs>     Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
Operation
CheckMOPSEnabled();

bits(64) toaddress = X[d];
bits(64) setsize = X[n];
bits(8) data = X[s];
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x007FFFFFFFFFFFF0<63:0>;
    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';
    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();
    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros();
    boolean zero_size_exceptions = MemSetZeroSizeCheck();
    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        if stage == MOPSSStage_Main then
            stagesetsize = setsize - postsize;
            if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                stagesetsize = postsize;
                if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            if supports_option_a then
                while SInt(stagesetsize) < 0 do
                    // IMP DEF selection of the block size that is worked on. While many
// implementations might make this constant, that is not assumed.
B = SETSizeChoice(toaddress, setsize, 1);
assert B <= -1 * SInt(stagesetsize);
Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
setsize = setsize + B;
stagesetsize = stagesetsize + B;
if stage != MOPSSstage_Prologue then
  X[n] = setsize;
else
  while UInt(stagesetsize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 1);
    assert B <= UInt(stagesetsize);
    Mem[toaddress, B, acctype] = Replicate(data, B);
    toaddress = toaddress + B;
    setsize = setsize - B;
    stagesetsize = stagesetsize - B;
    if stage != MOPSSstage_Prologue then
      X[n] = setsize;
      X[d] = toaddress;
  end while
  if stage == MOPSSstage_Prologue then
    X[n] = setsize;
    X[d] = toaddress;
end if
SETPT, SETMT, SETET

Memory Set, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and performs an implementation defined amount of the memory set. SETMT performs an implementation defined amount of the memory set. SETET performs the last part of the memory set.

Note

The inclusion of implementation defined amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is implementation defined.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPT, option A (which results in encoding PSTATE.C = 0):
• If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an implementation defined number of bytes set.
• PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

After execution of SETPT, option B (which results in encoding PSTATE.C = 1):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xd holds the original Xd + an implementation defined number of bytes set.
• Xn holds the saturated Xn - an implementation defined number of bytes set.
• PSTATE.\{N,Z,V\} are set to \{0,0,0\}.

For SETMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• Xn holds -1* number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  ◦ the value of Xn is written back with 0.
  ◦ the value of Xd is written back with the lowest address that has not been set.

For SETET, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• Xn holds -1* the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For SETET, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  ◦ the value of Xn is written back with 0.
  ◦ the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)
Epilogue (op2 == 1001)

SETET [<Xd>], <Xn>, <Xs>

Main (op2 == 0101)

SETMT [<Xd>], <Xn>, <Xs>

Prologue (op2 == 0001)

SETPT [<Xd>], <Xn>, <Xs>

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
MOPSSstage stage;
case op2<3:2> of
   when '00' stage = MOPSSStage_Prologue;
   when '01' stage = MOPSSStage_Main;
   when '10' stage = MOPSSStage_Epilogue;
   otherwise UNDEFINED;
if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
Operation
CheckMOPSEnabled();

bits(64) toaddress = X[d];
bits(64) setsize = X[n];
bits(8) data = X[s];
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSSStage_Prologue then
  if setsize<63> == '1' then setsize = 0x007FFFFFFFFFFFF0<63:0>;
  if supports_option_a then
    PSTATE.C = '0';
    toaddress = toaddress + setsize;
    setsize = Zeros(64) - setsize;
  else
    PSTATE.C = '1';
    PSTATE.N = '0';
    PSTATE.V = '0';
    PSTATE.Z = '0';

  stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
  assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();
  if SInt(setsize) > 0 then
    assert SInt(stagesetsize) <= SInt(setsize);
  else
    assert SInt(stagesetsize) >= SInt(setsize);

  else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros();
    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
      if supports_option_a then
        if PSTATE.C == '1' then
          boolean wrong_option = TRUE;
          boolean from_epilogue = stage == MOPSSStage_Epilogue;
          MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        else
          if PSTATE.C == '0' then
            boolean wrong_option = TRUE;
            boolean from_epilogue = stage == MOPSSStage_Epilogue;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
      if stage == MOPSSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
          boolean wrong_option = FALSE;
          boolean from_epilogue = FALSE;
          MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
        else
          stagesetsize = postsize;
          if setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

      if supports_option_a then
        while SInt(stagesetsize) < 0 do
          // IMP DEF selection of the block size that is worked on. While many
// implementations might make this constant, that is not assumed.
B = \textbf{SETSizeChoice}(toaddress, setsize, l);
assert B <= -1 \times \textbf{SInt}(\text{stagesetsize});

\text{Mem}[\text{toaddress}+\text{setsize}, B, \text{acctype}] = \textbf{Replicate}(\text{data}, B);
\text{setsize} = \text{setsize} + B;
\text{stagesetsize} = \text{stagesetsize} + B;
\text{if} \text{ stage } \neq \textbf{MOPSStage Prologue} \text{ then }
  X[n] = \text{setsize};
\text{else}
  \text{while} \ \textbf{UInt}(\text{stagesetsize}) > 0 \text{ do }
  \text{ // IMP DEF selection of the block size that is worked on. While many }
  \text{ // implementations might make this constant, that is not assumed. }
  B = \textbf{SETSizeChoice}(\text{toaddress}, \text{setsize}, 1);
  \text{assert } B \leq \textbf{UInt}(\text{stagesetsize});

  \text{Mem}[\text{toaddress}, B, \text{acctype}] = \textbf{Replicate}(\text{data}, B);
  \text{toaddress} = \text{toaddress} + B;
  \text{setsize} = \text{setsize} - B;
  \text{stagesetsize} = \text{stagesetsize} - B;
  \text{if} \text{ stage } \neq \textbf{MOPSStage Prologue} \text{ then }
    X[n] = \text{setsize};
    X[d] = \text{toaddress};
\text{if} \text{ stage } = \textbf{MOPSStage Prologue} \text{ then }
  X[n] = \text{setsize};
  X[d] = \text{toaddress};
SETPTN, SETMTN, SETETN

Memory Set, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPTN, option A (which results in encoding PSTATE.C = 0):
• If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xd holds the original Xd + saturated Xn.
• Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
• PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPTN, option B (which results in encoding PSTATE.C = 1):
• If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
• Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
• Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
• PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• Xn holds -1* number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  ◦ the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  ◦ the value of Xd is written back with the lowest address that has not been set.

For SETETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:
• Xn is treated as a signed 64-bit number.
• Xn holds -1* the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to -Xn.
• At the end of the instruction, the value of Xn is written back with 0.

For SETETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:
• Xn holds the number of bytes remaining to be set in the memory set in total.
• Xd holds the lowest address that the set is made to.
• At the end of the instruction:
  ◦ the value of Xn is written back with 0.
  ◦ the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

op2
Epilogue (op2 == 1011)

SETETN [<Xd>]!, <Xn>!, <Xs>

Main (op2 == 0111)

SETMTN [<Xd>]!, <Xn>!, <Xs>

Prologue (op2 == 0011)

SETPTN [<Xd>]!, <Xn>!, <Xs>

if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;
integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
MOPSStage stage;
case op2<3:2> of
  when '00' stage = MOPSStage_Prologue;
  when '01' stage = MOPSStage_Main;
  when '10' stage = MOPSStage_Epilogue;
otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.
Operation
CheckMOPSEnabled();

bits(64) toaddress = X[d];
bite(64) setsize = X[n];
bite(8) data = X[s];
bite(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x007FFFFFFFFFFFF0<63:0>;

    if supports_option_a then
        PSTATE.C = '0';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        PSTATE.C = '1';
        PSTATE.N = '0';
        PSTATE.V = '0';
        PSTATE.Z = '0';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros();

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
    
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros();

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if PSTATE.C == '1' then
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                if PSTATE.C == '0' then
                    boolean wrong_option = TRUE;
                    boolean from_epilogue = stage == MOPSSStage_Epilogue;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

        if stage == MOPSSStage_Main then
            stagesetsize = setsize - postsize;
            if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
                boolean wrong_option = FALSE;
                boolean from_epilogue = FALSE;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);
            else
                stagesetsize = postsize;
                if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
                    boolean wrong_option = FALSE;
                    boolean from_epilogue = TRUE;
                    MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, is_setg);

        if supports_option_a then
            while SInt(stagesetsize) < 0 do
                // IMP DEF selection of the block size that is worked on. While many
// implementations might make this constant, that is not assumed.
B = SETSizeChoice(toaddress, setsize, 1);
assert B <= -1 * SInt(stagesetsize);
Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
setsize = setsize + B;
stagesetsize = stagesetsize + B;
if stage != MOPSSStage_Prologue then
  X[n] = setsize;
else
  while UInt(stagesetsize) > 0 do
    // IMP DEF selection of the block size that is worked on. While many
    // implementations might make this constant, that is not assumed.
    B = SETSizeChoice(toaddress, setsize, 1);
    assert B <= UInt(stagesetsize);
    Mem[toaddress, B, acctype] = Replicate(data, B);
    toaddress = toaddress + B;
    setsize = setsize - B;
    stagesetsize = stagesetsize - B;
    if stage != MOPSSStage_Prologue then
      X[n] = setsize;
      X[d] = toaddress;
    end
  end
if stage == MOPSSStage_Prologue then
  X[n] = setsize;
  X[d] = toaddress;

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event*.

```
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 0 1 1 1 1
```

// Empty.

**Operation**

```
SendEvent();
```
SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

```
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 1 0 1 1 1 1 1
```

SEVL

// Empty.

Operation

```
SendEventLocal();
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | Rm | 0 | Ra | Rn | Rd |

SMADDL <Xd>, <Wn>, <Wm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d] = result<63:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Secure Monitor Call causes an exception to EL3. SMC is available only for software executing at EL1 or higher. It is **UNDEFINED** in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value 0x17, that is taken to EL3.

If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is **UNDEFINED**.

```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |

SMC `<imm>`

// Empty.

**Assembler Symbols**

`
<imm>` Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

**Operation**

```
AArch64.CheckForSMCUndefOrTrap(imm16);
AArch64.CallSecureMonitor(imm16);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of **SMSUBL**. This means:

- The encodings in this description are named to match the encodings of **SMSUBL**.
- The description of **SMSUBL** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Nm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**SMNEGL** \(<Xd>, <Wn>, <Wm>\)

is equivalent to

**SMSUBL** \(<Xd>, <Wn>, <Wm>, XZR\)

and is always the preferred disassembly.

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of **SMSUBL** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Xa>`: Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMNEGL</td>
<td>Ra == ‘1111’</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

integer result;
result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d] = result<63:0>;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMULH**

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | Rm | 0  | (1) (1) (1) (1) | Rn | Rd |

**SMULH** <Xd>, <Xn>, < Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

**Assembler Symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d] = result<127:64>;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 1 1 0 1 1 0 0 1 Rm 0 1 1 1 1 1 Rn Rd
```

SMULL <Xd>, <Wn>, <Wm>

is equivalent to

SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

• When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  ◦ The store is to the same location as the load.
  ◦ The store uses the same virtual address as the load.
  ◦ The store appears in program order before the SSBB.

• When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  ◦ The store is to the same location as the load.
  ◦ The store uses the same virtual address as the load.
  ◦ The store appears in program order after the SSBB.

This is an alias of DSB. This means:

• The encodings in this description are named to match the encodings of DSB.
• The description of DSB gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>CRm</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SSBB is equivalent to

DSB #0

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.
ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index
(FEAT_MTE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

Pre-index
(FEAT_MTE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

Signed offset
(FEAT_MTE)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

Assembler Symbols

<Xt|SP> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
ST64B

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

Integer
(FEATURE_LS64)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Rn  Rt

ST64B <Xt>, [<Xn|SP> {},#0]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

MemStore64B(address, data, acctype);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST64BV

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer
(FEAT_LS64_V)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|:---:|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | Rs | 1 | 0 | 1 | 1 | 0 | 0 | Rn | -- | -- | -- | -- | -- | -- | Rt |

ST64BV <Xs>, <Xt>, [<Xn|SP>]

if !HaveFeatLS64_V() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;

Assembler Symbols

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.
The value returned is:
0xFFFFFFFF_FFFFFFFF
If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= 0xFFFFFFFF_FFFFFFFF
If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

`CheckST64BVEnabled();`

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = `AccType_ATOMICLS64`;

if `HaveMTE2Ext()` then
    `SetTagCheckedInstruction(tag_checked);`

for i = 0 to 7
    value = `X[t+i];`
    if `BigEndian(acctype)` then value = `BigEndianReverse(value);`
    data<63+64*i:64*i> = value;

if n == 31 then
    `CheckSPAlignment();`
else
    address = `SP[ ];`
status = `MemStore64BWithRet(address, data, acctype);`

if s != 31 then `X[s]` = status;
ST64BV0

Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, with the bottom 32 bits taken from ACCDATA_EL1, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer
(FEAT_LS64_ACCDATA)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

ST64BV0 <Xs>, <Xt>, [<Xn|SP>]

if !HaveFeatLS64_ACCDATA() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;

Assembler Symbols

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.
The value returned is:
0xFFFFFFFF_FFFFFFFF
    If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.
!= 0xFFFFFFFF_FFFFFFFF
    If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

\texttt{CheckST64BV0Enabled();}

\texttt{bits(512) data;}
\texttt{bits(64) address;}
\texttt{bits(64) value;}
\texttt{bits(64) status;}
\texttt{acctype = AccType_ATOMICLS64;}

\texttt{if HaveMTE2Ext() then}
\texttt{	SetTagCheckedInstruction(tag_checked);}

\texttt{bits(64) Xt = X[t];}
\texttt{value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;;}
\texttt{if BigEndian(acctype) then value = BigEndianReverse(value);
if n == 31 then CheckSPAlignment();
else address = X[n];}
\texttt{status = MemStore64BWithRet(address, data, acctype);}
\texttt{if s != 31 then X[s] = status;
STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD does not have release semantics.
- STADDL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

### Integer

(Feat_LSE)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | x   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | R   | 1   | Rs  | 0   | 0   | 0   | 0   | 0   | Rn  | 1   | 1   | 1   | 1   | 1   | 1   | 0   | 0   | R   | 1   | Rs  |

#### 32-bit LDADD alias (size == 10 && R == 0)

STADD <Ws>, [<Xn|SP>]

is equivalent to

LDADD <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDADDL alias (size == 10 && R == 1)

STADDL <Ws>, [<Xn|SP>]

is equivalent to

LDADDL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDADD alias (size == 11 && R == 0)

STADD <Xs>, [<Xn|SP>]

is equivalent to

LDADD <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDADDL alias (size == 11 && R == 1)

STADDL <Xs>, [<Xn|SP>]

is equivalent to

LDADDL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL, gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB does not have release semantics.
- STADDLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

### Integer

(FEAT_LSE)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | A | opc | Rn | Rt |

**No memory ordering (R == 0)**

STADDB <Ws>, [<Xn|SP>]

is equivalent to

LDADDB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STADDLB <Ws>, [<Xn|SP>]

is equivalent to

LDADDB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH does not have release semantics.
- STADDLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 1 1 1 1 0 0 0 0 R 1 | Rs 0 0 0 0 0 0 Rn 1 1 1 1 1 |

size A opc Rn Rt

No memory ordering (R == 0)

STADDH <Ws>, [<Xn|SP>]

is equivalent to

LDADDH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDLH <Ws>, [<Xn|SP>]

is equivalent to

LDADDLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit LDCLR alias (size == 10 && R == 0)

STCLR <Ws>, [<Xn|SP>]

is equivalent to

LDCLR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

32-bit LDCLRL alias (size == 10 && R == 1)

STCLRL <Ws>, [<Xn|SP>]

is equivalent to

LDCLRL <Ws>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDCLR alias (size == 11 && R == 0)

STCLR <Xs>, [<Xn|SP>]

is equivalent to

LDCLR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDCLRL alias (size == 11 && R == 1)

STCLRL <Xs>, [<Xn|SP>]

is equivalent to

LDCLRL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

### Operational Information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB does not have release semantics.
- STCLRLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLRBB, LDCLRAB, LDCLRALB, LDCLRLB. This means:

- The encodings in this description are named to match the encodings of LDCLRBB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRBB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Integer (FEAT_LSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0  0  1  1  1  0  0  0  0  R  1  Rs  0  0  0  1  0  0  Rn  1  1  1  1  1</td>
</tr>
<tr>
<td>size  A  opc  Rn  Rt</td>
</tr>
</tbody>
</table>

No memory ordering (R == 0)

STCLRB <Ws>, [<Xn|SP>]

is equivalent to

LDCLRBB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STCLRLB <Ws>, [<Xn|SP>]

is equivalent to

LDCLRLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRBB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH does not have release semantics.
- STCLRLH stores to memory with release semantics, as described in `Load-Acquire, Store-Release`.

For information about memory accesses see `Load/Store addressing modes`.

This is an alias of `LDCLRH, LDCLRRAH, LDCLRHALH, LDCLRRLH`. This means:

- The encodings in this description are named to match the encodings of `LDCLRH, LDCLRRAH, LDCLRHALH, LDCLRRLH`.
- The description of `LDCLRH, LDCLRRAH, LDCLRHALH, LDCLRRLH` gives the operational pseudocode for this instruction.

**Integer**

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size A opc Rt
```

**No memory ordering (R == 0)**

STCLRH `<Ws>`, `[<Xn|SP>]`

is equivalent to

LDCLRH `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**Release (R == 1)**

STCLRLH `<Ws>`, `[<Xn|SP>]`

is equivalent to

LDCLRRLH `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Ws>`: Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of `LDCLRH, LDCLRRAH, LDCLRHALH, LDCLRRLH` gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR does not have release semantics.
- STEORL stores to memory with release semantics, as described in `Load-Acquire, Store-Release`.

For information about memory accesses see `Load/Store addressing modes`.

This is an alias of `LDEOR, LDEORA, LDEORAL, LDEORL`. This means:

- The encodings in this description are named to match the encodings of `LDEOR, LDEORA, LDEORAL, LDEORL`.
- The description of `LDEOR, LDEORA, LDEORAL, LDEORL` gives the operational pseudocode for this instruction.

### Integer

(FEAT_LSE)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------------------------|---------------------------------|
| size  | A                | opc               | Rt                |
| 1 x   | 1 1 1 0 0 0 0    | R 1               | Rs                |
| 0 1 0 | 0 0              | 1 0 0             | 0 0               | Rn                |
| 1 1 1 | 1 1 1 1 1 1 1    |                   |                   |                   |
```

**32-bit LDEOR alias (size == 10 && R == 0)**

STEOR `<Ws>`, `[<Xn|SP>]`

is equivalent to

LDEOR `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**32-bit LDEORL alias (size == 10 && R == 1)**

STEORL `<Ws>`, `[<Xn|SP>]`

is equivalent to

LDEORL `<Ws>`, WZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**64-bit LDEOR alias (size == 11 && R == 0)**

STEOR `<Xs>`, `[<Xn|SP>]`

is equivalent to

LDEOR `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.

**64-bit LDEORL alias (size == 11 && R == 1)**

STEORL `<Xs>`, `[<Xn|SP>]`

is equivalent to

LDEORL `<Xs>`, XZR, `[<Xn|SP>]`

and is always the preferred disassembly.
**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STEORB, STEORLB**

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB does not have release semantics.
- STEORLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

### Integer

(Feat_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | 1  | Rt |

**No memory ordering (R == 0)**

STEORB <Ws>, [Xn|SP]

is equivalent to

LDEORB <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

**Release (R == 1)**

STEORLB <Ws>, [Xn|SP]

is equivalent to

LDEORLB <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH does not have release semantics.
- STEORLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1 |

size A opc Rn Rt

No memory ordering (R == 0)

STEORH <Ws>, [<Xn|SP>]

is equivalent to

LDEORH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STEORLH <Ws>, [<Xn|SP>]

is equivalent to

LDEORLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STG

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

### Post-index

(Feat_MTE)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 1 0 0 1 0 0 1 imm9 0 1 Xn Xt
```

**STG <Xt|SP>, [<Xn|SP>], #<simm>**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bis(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

### Pre-index

(Feat_MTE)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 1 0 0 1 1 1 imm9 1 1 Xn Xt
```

**STG <Xt|SP>, [<Xn|SP>, #<simm>]**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bis(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

### Signed offset

(Feat_MTE)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 1 0 0 1 imm9 1 0 Xn Xt
```

**STG <Xt|SP>, [<Xn|SP>{, #<simm>}]**

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bis(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

### Assembler Symbols

*Xt*|SP* Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is undefined at EL0.

This instruction generates an Unchecked access.

Integer (FEAT_MTE2)

STGM <Xt>, [<Xn|SP>]

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = data<(index*4)+3:index*4>;
    AArch64_MemTag[address, AccType_NORMAL] = tag;
    address = address + TAG_GRANULE;
    index = index + 1;
STGP

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index
(FEAT_MTE)

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \text{simm7} & \text{Xt2} & \text{Xn} & \text{Xt} \\
\end{array}
\]

STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bias(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

Pre-index
(FEAT_MTE)

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \text{simm7} & \text{Xt2} & \text{Xn} & \text{Xt} \\
\end{array}
\]

STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>!

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bias(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

Signed offset
(FEAT_MTE)

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{simm7} & \text{Xt2} & \text{Xn} & \text{Xt} \\
\end{array}
\]

STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm}>]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bias(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

Assembler Symbols

<xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2>  Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<imm>  For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.

For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

**Operation**

```plaintext
bits(64) address;
b bits(64) data1;
b bits(64) data2;

SetTagCheckedInstruction(FALSE);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data1 = X[t];
data2 = X[t2];

if !postindex then
  address = address + offset;

if address != Align(address, TAG_GRANULE) then
  AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);

if writeback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset
(FEAT_LOR)

<table>
<thead>
<tr>
<th>1</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>0</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L</td>
<td>Rs</td>
<td>o0</td>
<td>Rt2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

STLLR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLLR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];
data = X[t];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

### No offset

**(FEAT_LOR)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |
```

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rs</th>
<th>o0</th>
<th>Rt</th>
</tr>
</thead>
</table>

**STLLRB** `<Wt>`, `<Xn|SP>{,#0}`

```plaintext```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```plaintext```

```
bits(64) address;
bites(8) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, 1, AccType_LIMITEDORDERED] = data;
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

**No offset**

*(FEAT_LOR)*

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 1 0 0 0 1 0 0 | (1) (1) (1) (1) (1) | 0 | (1) (1) (1) (1) (1) |
| size | L | Rs | o0 | Rt |

**STLLRH**

\[<Wt>, [<Xn|SP>\{,#0}\]

integer n = \text{UInt}(Rn);
integer t = \text{UInt}(Rt);
boolean tag_checked = n != 31;

**Assembler Symbols**

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

\[ \text{bits}(64) \text{ address;} \]
\[ \text{bits}(16) \text{ data;} \]
if \(\text{HaveMTE2Ext}()\) then
\quad \text{SetTagCheckedInstruction}(tag\_checked);
if n == 31 then
\quad \text{CheckSPAlignment}();
\quad \text{address} = \text{SP}[];
else
\quad \text{address} = \text{X}[n];
data = \text{X}[t];
\text{Mem}\{\text{address}, 2, \text{AccType\_LIMITEDORDERED}\} = \text{data};

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 x 0 0 1 0 0 0 | 1 0 0 1 1 | 1 0 1 1 (1) | 1 (1) (1) | 1 1 (1) | (1) (1) (1) (1) | Rn | Rt |
| size | L | Rs | o0 | Rt2 |

32-bit (size == 10)

STLR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLR <Xt>, [<Xn|SP>{,#0}]

| integer n = UInt(Rn); |
| integer t = UInt(Rt); |
| integer elsize = 8 < UInt(size); |
| boolean tag_checked = n != 31; |

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

| bits(64) address; |
| bits(elsize) data; |
| constant integer dbytes = elsize DIV 8; |
| if HaveMTE2Ext() then |
| SetTagCheckedInstruction(tag_checked); |
| if n == 31 then |
| CheckSPAlignment(); |
| address = SP[]; |
| else |
| address = X[n]; |
| data = X[t]; |
| Mem[address, dbytes, AccType_ORDERED] = data; |

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STLRB**

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 1   | 0   | 0   | 0   | 1   | 0   | 0   | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | Rn  | Rs  | o0  | Rt2 |

**STLRB** `<Wt>`, `<Xn|SP>{,#0}`

integer \( n = \text{UInt}(Rn); \)

integer \( t = \text{UInt}(Rt); \)

boolean tag_checked = \( n \neq 31; \)

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

bits(64) address;

bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if \( n == 31 \) then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = X[t];

Mem[address, 1, AccType_ORDERED] = data;

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, 2, AccType_ORDERED] = data;
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about memory accesses, see Load/Store addressing modes.

### Unscaled offset

(FeaT_LRCPC2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

STLUR <Wt>, [<Xn|SP>{{, #<simm>}}]

64-bit (size == 11)

STLUR <Xt>, [<Xn|SP>{{, #<simm>}}]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

### Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = n != 31;

### Operation

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_ORDERED] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(FeaT_LRCPC2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | imm9 | 0  | 0  | Rn  | 0  | 0  | Rt  |

STLURB <Wt>, [<Xn|SP>\}, #<simm>]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;

data = X[t];
Mem[address, 1, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>Rt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>{, #<simm>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(16) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
data = X[t];
Mem[address, 2, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

32-bit (sz == 0)

STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs); // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE; // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn unknown = TRUE; // address is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly STLXP.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(AccType_ORDEREDATOMIC) then el1:el2 else el2:el1;
    bit status = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
        status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);
```
**STLXR**

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

### 32-bit (size == 10)

STLXR <Ws>, <Wt>, [<Xn|SP>{,,#0}]

### 64-bit (size == 11)

STLXR <Ws>, <Xt>, [<Xn|SP>{,,#0}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);  // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

For information about the CONstrained UNpredictable behavior of this instruction, see *Architectural Constraints on UNpredictable behaviors*, and particularly *STLXR*.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:
- Memory is not updated.
Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize \div 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else if rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | (1) | (1) | (1) | (1) |    | Rn |    |    |    |    |    |    |
| size | L | 0 | 0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

STLXRB $<W_s>$, $<W_t>$, [<$X_n$]|SP>{,#0}]

integer $n =$ UInt(Rn);
integer $t =$ UInt(Rt);
integer $s =$ UInt(Rs);  // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

$<W_s>$ Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0 If the operation updates memory.
1 If the operation fails to update memory.

$<W_t>$ Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

$<X_n|$SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- $<W_s>$ is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | L | o0 | Rs | 1 | (1)(1)(1)(1)(1) | Rn | Rt |

STLXRH \(<Ws>, <Wt>, [<Xn|SP>{,#0}]\)

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    end case;
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    end case;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXRH*.

**Assembler Symbols**

\(<Ws>\) Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Ws>\) is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**STNP**

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.

<table>
<thead>
<tr>
<th>32-bit (opc == 00)</th>
<th>64-bit (opc == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STNP &lt;Wt1&gt;, &lt;Wt2&gt;, [&lt;Xn</td>
<td>SP&gt;{, #&lt;imm&gt;}]</td>
</tr>
</tbody>
</table>

// Empty.

### Assembler Symbols

- **<Wt1>** Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.
- **<Wt2>** Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.
- **<Xt1>** Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.
- **<Xt2>** Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.
- **<imm>** For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

### Shared Decode

```plaintext
type integer n = UInt(Rn);
type integer t = UInt(Rt);
type integer t2 = UInt(Rt2);
if opc<@> == '1' then UNDEFINED;
type integer scale = 2 + UInt(opc<1>);
type integer datasize = 8 << scale;
type bits(64) offset = LSL(SignExtend(imm7, 64), scale);
type boolean tag_checked = n != 31;
```
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data1 = X[t];
data2 = X[t2];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*. It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

### Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>x 0 1 0 1 0 0 0 1 0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (opc == 00)**

STP \(<Wt1>, <Wt2>, [<Xn|SP>], #<imm>\)

**64-bit (opc == 10)**

STP \(<Xt1>, <Xt2>, [<Xn|SP>], #<imm>\)

boolean wback = TRUE;
boolean postindex = TRUE;

### Pre-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>x 0 1 0 1 0 0 1 1 0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (opc == 00)**

STP \(<Wt1>, <Wt2>, [<Xn|SP>], #<imm>\)

**64-bit (opc == 10)**

STP \(<Xt1>, <Xt2>, [<Xn|SP>], #<imm>\)

boolean wback = TRUE;
boolean postindex = FALSE;

### Signed offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>x 0 1 0 1 0 0 1 0 0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (opc == 00)**

STP \(<Wt1>, <Wt2>, [<Xn|SP>], #<imm>\)

**64-bit (opc == 10)**

STP \(<Xt1>, <Xt2>, [<Xn|SP>], #<imm>\)

boolean wback = FALSE;
boolean postindex = FALSE;
For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

### Assembler Symbols

- `<Wt1>`: Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Wt2>`: Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xt1>`: Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>`: Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as `<imm>/4`.
  
  For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as `<imm>/4`.
  
  For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as `<imm>/8`.
  
  For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as `<imm>/8`.

### Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;  // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

STP
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2];
if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    if BigEndian(AccType_NORMAL) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    Mem[address, 2*dbytes, AccType_NORMAL, TRUE] = full_data;
else
    Mem[address, dbytes, AccType_NORMAL] = data1;
    Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

32-bit (size == 10)

\[
\text{STR}<Wt>, [<Xn|SP>], #<\text{simm}>
\]

64-bit (size == 11)

\[
\text{STR}<Xt>, [<Xn|SP>], #<\text{simm}>
\]

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = \text{UInt}(\text{size});
bits(64) offset = \text{SignExtend}(\text{imm9}, 64);

Pre-index

32-bit (size == 10)

\[
\text{STR}<Wt>, [<Xn|SP>], #<\text{simm}>
\]

64-bit (size == 11)

\[
\text{STR}<Xt>, [<Xn|SP>], #<\text{simm}>
\]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = \text{UInt}(\text{size});
bits(64) offset = \text{SignExtend}(\text{imm9}, 64);

Unsigned offset

32-bit (size == 10)

\[
\text{STR}<Wt>, [<Xn|SP>], #<\text{simm}>
\]

64-bit (size == 11)

\[
\text{STR}<Xt>, [<Xn|SP>], #<\text{simm}>
\]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = \text{UInt}(\text{size});
bits(64) offset = \text{SignExtend}(\text{imm9}, 64);
32-bit (size == 10)

STR <Wt>, [<Xn|SP>{{, #<pimm>}}]

64-bit (size == 11)

STR <Xt>, [<Xn|SP>{{, #<pimm>}}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
Operation

bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t];
    Mem[address, datasize DIV 8, AccType_NORMAL] = data;
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | Rm | option | S | 1 | 0 | Rn | Rt |

size opc

32-bit (size == 10)

STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>)], <extend> {<amount>}

64-bit (size == 11)

STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>)], <extend> {<amount>}

integer scale = UInt(size);
if option<0> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
**Shared Decode**

\[
\text{integer } n = \text{UInt}(Rn); \\
\text{integer } t = \text{UInt}(Rt); \\
\text{integer } m = \text{UInt}(Rm); \\
\text{integer } \text{datasize} = 8 \ll \text{scale};
\]

**Operation**

\[
\text{bits(64) offset} = \text{ExtendReg}(m, \text{extend_type}, \text{shift}); \\
\text{bits(64) address}; \\
\text{bits(datasize) data}; \\
\text{if HaveMTE2Ext() then} \\
\quad \text{SetTagCheckedInstruction}(\text{TRUE}); \\
\text{if } n == 31 \text{ then} \\
\quad \text{CheckSPAlignment}(); \\
\quad \text{address} = \text{SP}[]; \\
\text{else} \\
\quad \text{address} = X[n]; \\
\text{address} = \text{address} + \text{offset}; \\
\text{data} = X[t]; \\
\text{Mem}[_{\text{address}}, \text{datasize} \div \text{8}, \text{AccType_NORMAL}] = \text{data};
\]

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: *Post-index*, *Pre-index* and *Unsigned offset*.

Post-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|-----------------|-----------------|
| size                | opc             |
```

```java
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|-----------------|-----------------|
| size                | opc             |
```

```java
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|-----------------|-----------------|
| size                | opc             |
```

```java
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONstrained unPREDITABLE behavior of this instruction, see *Architectural Constraints on UNPREDITABLE behaviors*, and particularly STRB (immediate).

Assembler Symbols

- `<Wt>`  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`  Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>`  Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
integer n = Uint(Rn);
integer t = Uint(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
  c = ConstraitUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt_unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

if rt_unknown then
  data = bits(8) UNKNOWN;
else
  data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STRB (register)**

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------------------|----------------|-----------------------------|
| 0 0 1 1 1 0 0 0 0 1 | Rm | option | S 1 0 | Rn | Rt |

**Extended register (option != 011)**

\[
\text{STRB} \ <Wt>, [\langle Xn|SP\rangle, (\langle Wm\rangle|\langle Xm\rangle), \langle extend \rangle \{\langle amount \rangle\}]
\]

**Shifted register (option == 011)**

\[
\text{STRB} \ <Wt>, [\langle Xn|SP\rangle, \langle Xm\rangle\{, LSL \langle amount \rangle\}]
\]

if option<1> == '0' then UNDEFINED; // sub-word index

ExtendType extend_type = DecodeRegExtend(option);

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` Is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>` Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

**Shared Decode**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```
Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rtUnknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rtUnknown = TRUE;  // value stored is UNKNOWN
        when Constraint_UNDEF    UNDEFINED;
        when Constraint_NOP      EndOfInstruction();

Operation

bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if data = bits(16) UNKNOWN;
else
    data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size opc

STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>)], {<extend> {<amount>}}]

if option<1> == '0' then UNDEFINED; // sub-word index

ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = \texttt{ExtendReg}(m, \texttt{extend\_type}, shift);
bits(64) address;
bits(16) data;

if \texttt{HaveMTE2Ext}() then
    \texttt{SetTagCheckedInstruction}(\texttt{TRUE});

if \texttt{n} == 31 then
    \texttt{CheckSPAlignment}();
    address = \texttt{SP}[];
else
    address = \texttt{X}[n];

address = address + offset;

data = \texttt{X}[t];
\texttt{Mem[address, 2, AccType\_NORMAL]} = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- **STSET** does not have release semantics.
- **STSETL** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSET, LDSETA, LDSETAL, LDSETL**. This means:

- The encodings in this description are named to match the encodings of **LDSET, LDSETA, LDSETAL, LDSETL**.
- The description of **LDSET, LDSETA, LDSETAL, LDSETL** gives the operational pseudocode for this instruction.

### Integer

(\texttt{FEAT\_LSE})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 1  | 1  | 0  | 0  | 0  | R | 1  | Rs | 0  | 0 | 1 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 | size | A | opc | Rt |

**32-bit LDSET alias (size == 10 && R == 0)**

STSET <Ws>, [<Xn|SP>]

is equivalent to

LDSET <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**32-bit LDSETL alias (size == 10 && R == 1)**

STSETL <Ws>, [<Xn|SP>]

is equivalent to

LDSETL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDSET alias (size == 11 && R == 0)**

STSET <Xs>, [<Xn|SP>]

is equivalent to

LDSET <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDSETL alias (size == 11 && R == 1)**

STSETL <Xs>, [<Xn|SP>]

is equivalent to

LDSETL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB does not have release semantics.
- STSETLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size A opc Rn Rt

No memory ordering (R == 0)

STSETB <Ws>, [<Xn|SP>]

is equivalent to

LDSETB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSETLB <Ws>, [<Xn|SP>]

is equivalent to

LDSETLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a
bitwise OR with the value held in a register on it, and stores the result back to memory.

- **STSETH** does not have release semantics.
- **STSETLH** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSETH, LDSETAH, LDSETALH, LDSETLH**. This means:

- The encodings in this description are named to match the encodings of **LDSETH, LDSETAH, LDSETALH, LDSETLH**.
- The description of **LDSETH, LDSETAH, LDSETALH, LDSETLH** gives the operational pseudocode for this instruction.

**Integer**

*(FEAT_LSE)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | R | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | |

**No memory ordering (R == 0)**

**STSETH** <Ws>, [<Xn|SP>]

is equivalent to

**LDSETH** <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

**STSETLH** <Ws>, [<Xn|SP>]

is equivalent to

**LDSETLH** <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **LDSETH, LDSETAH, LDSETALH, LDSETLH** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSMAX, STSMAXL**

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL**. This means:

- The encodings in this description are named to match the encodings of **LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL**.
- The description of **LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL** gives the operational pseudocode for this instruction.

### Integer

(***FEAT_LSE***)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1 | Rs | 0 | 1 | 0 | 0 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | R | t |

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

#### 32-bit LDSMAX alias (size == 10 && R == 0)

**STSMAX** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDSMAX** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### 32-bit LDSMAXL alias (size == 10 && R == 1)

**STSMAXL** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDSMAXL** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDSMAX alias (size == 11 && R == 0)

**STSMAX** `<Xs>, [<Xn|SP>]`

is equivalent to

**LDSMAX** `<Xs>, XZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDSMAXL alias (size == 11 && R == 1)

**STSMAXL** `<Xs>, [<Xn|SP>]`

is equivalent to

**LDSMAXL** `<Xs>, XZR, [<Xn|SP>]`

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXB does not have release semantics.
- STSMAXLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size A opc Rt

No memory ordering (R == 0)

STSMAXB <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMAXLB <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSMAXH, STSMAXLH**

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXH does not have release semantics.
- STSMAXLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

### Integer
**(FEAT_LSE)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | 1  |

#### No memory ordering (R == 0)

STSMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### Release (R == 1)

STSMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN does not have release semantics.
- STSMINL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

### Integer (FEAT_LSE)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| 1  | x  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 |

- **size**
- **Rn**
- **Rt**
- **opc**
- **Rs**

### 32-bit LDSMIN alias (size == 10 && R == 0)

STSMIN <Ws>, [<Xn|SP>]

is equivalent to

LDSMIN <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 32-bit LDSMINL alias (size == 10 && R == 1)

STSMINL <Ws>, [<Xn|SP>]

is equivalent to

LDSMINL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDSMIN alias (size == 11 && R == 0)

STSMIN <Xs>, [<Xn|SP>]

is equivalent to

LDSMIN <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

### 64-bit LDSMINL alias (size == 11 && R == 1)

STSMINL <Xs>, [<Xn|SP>]

is equivalent to

LDSMINL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs>  Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- **STSMINB** does not have release semantics.
- **STSMINLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB**. This means:

- The encodings in this description are named to match the encodings of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB**.
- The description of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB** gives the operational pseudocode for this instruction.

### Integer

**FEAT_LSE**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

**size** | **A** | **opc** | **Rt**

**No memory ordering (R == 0)**

**STSMINB** <Ws>, [<Xn|SP>]

is equivalent to

**LDSMINB** <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

**STSMINLB** <Ws>, [<Xn|SP>]

is equivalent to

**LDSMINLB** <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINH does not have release semantics.
- STSMINLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

### Integer (FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size | A | opc | Rs |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rn | 1  | 1  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**No memory ordering (R == 0)**

STSMINH <Ws>, [<Xn|SP>]

is equivalent to

LDSMINH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STSMINLH <Ws>, [<Xn|SP>]

is equivalent to

LDSMINLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 0  | 0  | 0  | imm9| 1  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size| opc|
```

32-bit (size == 10)

STTR <Wt>, [<Xn|SP>, {, #simm}]

64-bit (size == 11)

STTR <Xt>, [<Xn|SP>, {, #simm}]

```
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 & & !(EL2Enabled() & & HaveNVExt() & & HCR_EL2.<NV,NV1> == '1');
unpriv_at_el2 = PSTATE.EL == EL2 & & HaveVirtHostExt() & & HCR_EL2.<E2H,TGE> == '1';

user_access_override = HaveUAOExt() & & PSTATE.UAO == '1';
if !user_access_override & & (unpriv_at_el1 || unpriv_at_el2) then
   acctype = AccType_UNPRIV;
else
   acctype = AccType_NORMAL;

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```
Operation

bits(64) address;
bets(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = X[t];
Mem[address, datasize DIV 8, acctype] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>imm9</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Rn</td>
</tr>
<tr>
<td>0</td>
<td>Rt</td>
</tr>
</tbody>
</table>

STTRB <Wt>, [<Xn|SP>{, #<simm>})

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !((EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '1'));
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '1';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = X[t];
Mem[address, 1, acctype] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | imm9 | 1  | 0  | Rn | Rt |

**STTRH** <Wt>, [<Xn|SP>{, #<simm}>]

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt“ field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn“ field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9“ field.

**Shared Decode**

```plaintext
tag_checked = n != 31;
```

**Operation**

```plaintext
address = address + offset;
data = X[t];
Mem[address, 2, acctype] = data;
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STUMAX, STUMAXL**

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- **STUMAX** does not have release semantics.
- **STUMAXL** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL**. This means:

- The encodings in this description are named to match the encodings of **LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL**.
- The description of **LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL** gives the operational pseudocode for this instruction.

**Integer (FEAT_LSE)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit LDUMAX alias (size == 10 && R == 0)**

**STUMAX** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDUMAX** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

**32-bit LDUMAXL alias (size == 10 && R == 1)**

**STUMAXL** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDUMAXL** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

**64-bit LDUMAX alias (size == 11 && R == 0)**

**STUMAX** `<Xs>, [<Xn|SP>]`

is equivalent to

**LDUMAX** `<Xs>, XZR, [<Xn|SP>]`

and is always the preferred disassembly.

**64-bit LDUMAXL alias (size == 11 && R == 1)**

**STUMAXL** `<Xs>, [<Xn|SP>]`

is equivalent to

**LDUMAXL** `<Xs>, XZR, [<Xn|SP>]`

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- **STUMAXB** does not have release semantics.
- **STUMAXLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB**. This means:

- The encodings in this description are named to match the encodings of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB**.
- The description of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB** gives the operational pseudocode for this instruction.

### Integer

(Feat_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **size**: A
- **opc**: Rs
- **Rt**: Rn

### No memory ordering (R == 0)

STUMAXB \(<Ws>, [Xn|SP]\>

is equivalent to

LDUMAXB \(<Ws>, WZR, [Xn|SP]\>

and is always the preferred disassembly.

### Release (R == 1)

STUMAXLB \(<Ws>, [Xn|SP]\>

is equivalent to

LDUMAXLB \(<Ws>, WZR, [Xn|SP]\>

and is always the preferred disassembly.

### Assembler Symbols

- \(<Ws>\) Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

### Integer
**FEAT_LSE**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size A opc Rt

**No memory ordering (R == 0)**

STUMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STUMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMIN does not have release semantics.
- STUMINL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

### Integer (FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |

#### 32-bit LDUMIN alias (size == 10 && R == 0)

STUMIN <Ws>, [<Xn|SP>]

is equivalent to

LDUMIN <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 32-bit LDUMINL alias (size == 10 && R == 1)

STUMINL <Ws>, [<Xn|SP>]

is equivalent to

LDUMINL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDUMIN alias (size == 11 && R == 0)

STUMIN <Xs>, [<Xn|SP>]

is equivalent to

LDUMIN <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

#### 64-bit LDUMINL alias (size == 11 && R == 1)

STUMINL <Xs>, [<Xn|SP>]

is equivalent to

LDUMINL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINI gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- **STUMINB** does not have release semantics.
- **STUMINLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB**. This means:

- The encodings in this description are named to match the encodings of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB**.
- The description of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB** gives the operational pseudocode for this instruction.

### Integer

*(FEAT_LSE)*

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td></td>
<td>Rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### No memory ordering (R == 0)

**STUMINB <Ws>, [<Xn|SP>]**

is equivalent to

**LDUMINB <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

#### Release (R == 1)

**STUMINLB <Ws>, [<Xn|SP>]**

is equivalent to

**LDUMINLB <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINH does not have release semantics.
- STUMINLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

size A opc Rn Rt

No memory ordering (R == 0)

STUMINH \(<Ws>\), \([<Xn|SP>]\)

is equivalent to

LDUMINH \(<Ws>\), WZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

Release (R == 1)

STUMINLH \(<Ws>\), \([<Xn|SP>]\)

is equivalent to

LDUMINLH \(<Ws>\), WZR, \([<Xn|SP>]\)

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (size == 10)

STUR <Wt>, [<Xn|SP>{, #<simm>}]  

64-bit (size == 11)

STUR <Xt>, [<Xn|SP>{, #<simm>}]  

integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);  

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.  
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.  

Shared Decode

integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer datasize = 8 << scale;  
boolean tag_checked = n != 31;  

Operation

bits(64) address;  
bits(datasize) data;  
if HaveMTE2Ext() then  
    SetTagCheckedInstruction(tag_checked);  
if n == 31 then  
    CheckSPAlignment();  
    address = SP[];  
else  
    address = X[n];  
address = address + offset;  
data = X[t];  
Mem[address, datasize DIV 8, AccType_NORMAL] = data;  

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STURB**

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see [Load/Store addressing modes](#).

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0 0 0</td>
<td>0 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

**Operation**

```java
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and
stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see Load/
Store addressing modes.

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in
  the "imm9" field.

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

### Operation

```
bits(64) address;
bites(16) data;

if HaveMTEx2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses, see Load/Store addressing modes.

### 32-bit (sz == 0)

STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,,#0}]

### 64-bit (sz == 1)

STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,,#0}]

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);  // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release
integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n & n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXP.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
  - 0 If the operation updates memory.
  - 1 If the operation fails to update memory.

- `<Xt1>` is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt2>` is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- `<Wt1>` is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:
  - Memory is not updated.
  - <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
  - If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
  - Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsiif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(datasize) UNKNOWN;
elsielse
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian(AccType_ATOMIC) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the physical memory locations corresponding to virtual address range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rs</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>o0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);  // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainsUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainsUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly STXR.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.
1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.
Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about memory accesses see Load/Store addressing modes.

For information about memory accesses see Load/Store addressing modes.

### Assembler Symbols

**<Ws>**

Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- **0** If the operation updates memory.
- **1** If the operation fails to update memory.

**<Wt>**

Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

**<Xn|SP>**

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- **<Ws>** is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly **STXRB**.
Operation

bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
endif

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];
endif

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STXRH**

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See [Synchronization and semaphores](#). The memory access is atomic.

For information about memory accesses see [Load/Store addressing modes](#).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

**STXRH** `<Ws>`, `<Wt>`, [`<Xn|SP>{,#0}`]

```plaintext
integer n = UInt(Rn);
ingen t = UInt(Rt);
ingen s = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

**Assembler Symbols**

- `<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
  - 0 If the operation updates memory.
  - 1 If the operation fails to update memory.

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:
- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

```c
bits(64) address;
bites(16) data;

if HaveMTEx2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index
(Feat_MTE)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & \text{imm9} & 0 & 1 & Xn & Xt \\
\end{array}
\]

STZ2G <Xt|SP>, [<Xn|SP>], #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

Pre-index
(Feat_MTE)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & \text{imm9} & 1 & 1 & Xn & Xt \\
\end{array}
\]

STZ2G <Xt|SP>, [<Xn|SP>], #<simm>]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

Signed offset
(Feat_MTE)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & \text{imm9} & 1 & 0 & Xn & Xt \\
\end{array}
\]

STZ2G <Xt|SP>, [<Xn|SP>{, #<simm}>}

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

Assembler Symbols

<Xt|SP> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));
Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;
if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

### Post-index (FEAT_MTE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | imm9 | 0  | 1  | Xn | Xt |

STZG `<Xt|SP>, [<Xn|SP>], #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

### Pre-index (FEAT_MTE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | imm9 | 1  | 1  | Xn | Xt |

STZG `<Xt|SP>, [<Xn|SP>, #<simm>]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

### Signed offset (FEAT_MTE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | imm9 | 1  | 0  | Xn | Xt |

STZG `<Xt|SP>, [<Xn|SP], #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

### Assembler Symbols

- `<Xt|SP>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- `<simm>` Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;

\texttt{SetTagCheckedInstruction}(FALSE);

if \( n == 31 \) then
  \texttt{CheckSPAlignment}();
  address = \texttt{SP}[];
else
  address = \texttt{X}[n];

if !postindex then
  address = address + offset;

if address != \texttt{Align}(address, \texttt{TAG\_GRANULE}) then
  \texttt{AArch64.Abort}(address, \texttt{AlignmentFault}(\texttt{AccType\_NORMAL}, \texttt{TRUE}, \texttt{FALSE}));

\texttt{Mem}[address, \texttt{TAG\_GRANULE}, \texttt{AccType\_NORMAL}] = \texttt{Zeros}(\texttt{TAG\_GRANULE} * 8);

bits(64) data = if \( t == 31 \) then \texttt{SP}[] else \texttt{X}[t];
bits(4) tag = \texttt{AArch64.AllocationTagFromAddress}(data);
\texttt{AArch64.MemTag}[address, \texttt{AccType\_NORMAL}] = tag;

if writeback then
  if postindex then
    address = address + offset;
  if \( n == 31 \) then
    \texttt{SP}[] = address;
  else
    \texttt{X}[n] = address;
STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag written to address A is taken from the source register bits<3:0>. This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

**Integer**

**(FEAT_MTE2)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

STZGM <Xt>, [<Xn|SP>]

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

**Operation**

if PSTATE.EL == EL0 then UNDEFINED;
bits(64) data = X[t];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
for i = 0 to count-1
    AArch64_MemTag[address, AccType_NORMAL] = tag;
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
    address = address + TAG_GRANULE;

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SUB (extended register)**

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | Rm | option | imm3 | Rn | Rd |

**32-bit (sf == 0)**

\[
\text{SUB } \langle \text{Wd}\,|\,\text{WSP} \rangle, \langle \text{Wn}\,|\,\text{WSP} \rangle, \langle \text{Wm} \rangle, \langle \text{extend} \rangle \{\langle \text{#amount} \rangle \}
\]

**64-bit (sf == 1)**

\[
\text{SUB } \langle \text{Xd}\,|\,\text{SP} \rangle, \langle \text{Xn}\,|\,\text{SP} \rangle, \langle \text{R} \rangle \langle \text{m} \rangle, \langle \text{extend} \rangle \{\langle \text{#amount} \rangle \}
\]

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( m = \text{UInt}(\text{Rm}) \);
integer datasize = if sf == '1' then 64 else 32;
\[
\text{ExtendType extend_type = DecodeRegExtend(option)};
\]
integer shift = \( \text{UInt}(\text{imm3}) \);
if shift > 4 then UNDEFINED;

**Assembler Symbols**

- \( \langle \text{Wd}\,|\,\text{WSP} \rangle \): Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \( \langle \text{Wn}\,|\,\text{WSP} \rangle \): Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \( \langle \text{Wm} \rangle \): Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \( \langle \text{Xd}\,|\,\text{SP} \rangle \): Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- \( \langle \text{Xn}\,|\,\text{SP} \rangle \): Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- \( \langle \text{R} \rangle \): Is a width specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>( \langle \text{R} \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

\( \langle m \rangle \): Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

\( \langle \text{extend} \rangle \): For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>( \langle \text{extend} \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>
If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUB (immediate)**

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);

**Assembler Symbols**

- `<Wd|WSP>` is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.
- `<Wn|WSP>` is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.
- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.
- `<Xn|SP>` is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.
- `<imm>` is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.
- `<shift>` is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Operation**

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
### SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias **NEG (shifted register)**.

#### 32-bit (sf == 0)

Subtract \( <Wd> \), \( <Wn> \), \( <Wm> \)}, \( <\text{shift}> \) \(#<\text{amount}>\)

#### 64-bit (sf == 1)

Subtract \( <Xd> \), \( <Xn> \), \( <Xm> \)}, \( <\text{shift}> \) \(#<\text{amount}>\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

#### Assembler Symbols

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<shift>** Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<amount>** For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

#### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG (shifted register)</td>
<td>( Rn == '11111' )</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

**Assembler Symbols**

- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Xd” field.
- `<Xn|SP>` is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Xn” field.
- `<uimm6>` is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the “uimm6” field.
- `<uimm4>` is an unsigned immediate, in the range 0 to 15, encoded in the “uimm4” field.

**Operation**

```plaintext
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';
(result, -) = AddWithCarry(operand1, NOT(offset), '1');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```
SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

Integer

\[ \text{SUBP} \; <Xd>, \; <Xn|SP>, \; <Xm|SP> \]

if \(!\text{HaveMTEExt}()\) then UNDEFINED;
integer \(d = \text{UInt}(Xd);\)
integer \(n = \text{UInt}(Xn);\)
integer \(m = \text{UInt}(Xm);\)

Assembler Symbols

\(<Xd>\) is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
\(<Xn|SP>\) is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
\(<Xm|SP>\) is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

bits(64) \(\text{operand1} = \text{if } n == 31 \text{ then } SP[] \text{ else } X[n];\)
bites(64) \(\text{operand2} = \text{if } m == 31 \text{ then } SP[] \text{ else } X[m];\)
\(\text{operand1} = \text{SignExtend}(\text{operand1}<55:0>, 64);\)
\(\text{operand2} = \text{SignExtend}(\text{operand2}<55:0>, 64);\)
bites(64) \(\text{result};\)
\(\text{operand2} = \text{NOT}(\text{operand2});\)
\((\text{result}, \text{ -}) = \text{AddWithCarry}(\text{operand1}, \text{operand2}, '1');\)
\(X[d] = \text{result};\)
**SUBPS**

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias **CMPP**.

### Integer (FEAT_MTE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SUBPS** `<Xd>`, `<Xn|SP>`, `<Xm|SP>`

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

### Assembler Symbols

- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm|SP>` Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPP</td>
<td>$ S == '1' &amp;&amp; Xd == '11111'$</td>
</tr>
</tbody>
</table>

### Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);
bits(64) result;
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```
**SUBS (extended register)**

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias **CMP (extended register)**.

### 32-bit (sf == 0)

```plaintext
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

### 64-bit (sf == 1)

```plaintext
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- `<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

- `<extend>` For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>
If “Rn” is ‘11111’ (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when “imm3” is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

### Operational information

- If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMP (immediate)**.

### 32-bit (sf == 0)

**SUBS**<sup>Wd>, <Wn|WSP>, #<imm>{, <shift>}

### 64-bit (sf == 1)

**SUBS**<sup>Xd>, <Xn|SP>, #<imm>{, <shift>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

```plaintext
case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Wd&gt;</td>
<td>Is the 32-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Wn</td>
<td>WSP&gt;</td>
</tr>
<tr>
<td>&lt;Xd&gt;</td>
<td>Is the 64-bit name of the general-purpose destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>&lt;Xn</td>
<td>SP&gt;</td>
</tr>
<tr>
<td>&lt;imm&gt;</td>
<td>Is an unsigned immediate, in the range 0 to 4095, encoded in the &quot;imm12&quot; field.</td>
</tr>
<tr>
<td>&lt;shift&gt;</td>
<td>Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in &quot;sh&quot;:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP (immediate)</strong></td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(4) nzcv;
operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBS (shifted register)**

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases **CMP (shifted register)** and **NEGS**.

32-bit $(sf == 0)$

```plaintext
SUBS <Wd>, <Wn>, <Wm>{{, <shift> #<amount>}}
```

64-bit $(sf == 1)$

```plaintext
SUBS <Xd>, <Xn>, <Xm>{{, <shift> #<amount>}}
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
 ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

**Assembler Symbols**

<wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP (shifted register)</strong></td>
<td>Rd == '11111'</td>
</tr>
<tr>
<td><strong>NEGS</strong></td>
<td>Rn == '11111' &amp;&amp; Rd != '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SVC

Supervisor Call causes an exception to be taken to EL1. On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in `ESR_ELx`, using the EC value 0x15, and the value of the immediate argument.

```
  1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

SVC `#<imm>`

// Empty.

Assembler Symbols

`<imm>` Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSVCTrap(imm16);
AArch64.CallSupervisor(imm16);
```
SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Aquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

### Integer

*(FEAT_LSE)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size
32-bit SWP (size == 10 && A == 0 && R == 0)
SWP <Ws>, <Wt>, [<Xn|SP]>

32-bit SWPA (size == 10 && A == 1 && R == 0)
SWPA <Ws>, <Wt>, [<Xn|SP]>

32-bit SWPAL (size == 10 && A == 1 && R == 1)
SWPAL <Ws>, <Wt>, [<Xn|SP]>

32-bit SWPL (size == 10 && A == 0 && R == 1)
SWPL <Ws>, <Wt>, [<Xn|SP]>

64-bit SWP (size == 11 && A == 0 && R == 0)
SWP <Xs>, <Xt>, [<Xn|SP]>

64-bit SWPA (size == 11 && A == 1 && R == 0)
SWPA <Xs>, <Xt>, [<Xn|SP]>

64-bit SWPAL (size == 11 && A == 1 && R == 1)
SWPAL <Xs>, <Xt>, [<Xn|SP]>

64-bit SWPL (size == 11 && A == 0 && R == 1)
SWPL <Xs>, <Xt>, [<Xn|SP]>

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[0];
else
    address = X[n];
endif

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, regsize);
```
SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

### Integer (FEAT_LSE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SWPAB (A == 1 & R == 0)**

\[ \text{SWPAB} <Ws>, <Wt>, [<Xn|SP>] \]

**SWPALB (A == 1 & R == 1)**

\[ \text{SWPALB} <Ws>, <Wt>, [<Xn|SP>] \]

**SWPB (A == 0 & R == 0)**

\[ \text{SWPB} <Ws>, <Wt>, [<Xn|SP>] \]

**SWPLB (A == 0 & R == 1)**

\[ \text{SWPLB} <Ws>, <Wt>, [<Xn|SP>] \]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

\[ \text{AccType ldacctype} = \begin{cases} \text{AccType_ORDEREDATOMICRW} & \text{if } A == '1' \text{ & } Rt != '11111' \\ \text{AccType_ATOMICRW} & \text{else} \end{cases} \]

\[ \text{AccType stacctype} = \begin{cases} \text{AccType_ORDEREDATOMICRW} & \text{if } R == '1' \\ \text{AccType_ATOMICRW} & \text{else} \end{cases} \]

boolean tag_checked = n != 31;

**Assembler Symbols**

- <Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bites(8) data;
bites(8) store_value;

if HaveMTE2Ex() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

```
Integer
(FEAT_LSE)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | Rn | Rt | size
```

**SWPAH (A == 1 && R == 0)**

SWPAH <Ws>, <Wt>, [<Xn|SP>]

**SWPALH (A == 1 && R == 1)**

SWPALH <Ws>, <Wt>, [<Xn|SP>]

**SWPH (A == 0 && R == 0)**

SWPH <Ws>, <Wt>, [<Xn|SP>]

**SWPLH (A == 0 && R == 1)**

SWPLH <Ws>, <Wt>, [<Xn|SP>]

```java
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(16) data;
bits(16) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 & N == 0)

SXTB <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

64-bit (sf == 1 & N == 1)

SXTB <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

### 32-bit (sf == 0 & N == 0)

SXTH \(<Wd>, <Wn>\)

is equivalent to

SBFM \(<Wd>, <Wn>, \#0, \#15\)

and is always the preferred disassembly.

### 64-bit (sf == 1 & N == 1)

SXTH \(<Xd>, <Wn>\)

is equivalent to

SBFM \(<Xd>, <Xn>, \#0, \#15\)

and is always the preferred disassembly.

**Assembler Symbols**

\(<Wd>\)  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xd>\)  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xn>\)  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<Wn>\)  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of SBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

---

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

64-bit

SXTW \(<Xd>, <Wn>\)

is equivalent to

SBFM \(<Xd>, <Xn>, #0, #31\)

and is always the preferred disassembly.

Assembler Symbols

- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Wn>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SYS

System instruction. For more information, see Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.

This instruction is used by the aliases AT, CFP, CPP, DC, DVP, IC, and TLBI.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

Assembler Symbols

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>CRn == '0111' &amp;&amp; CRm == '100x' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_AT</td>
</tr>
<tr>
<td>CFP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '100'</td>
</tr>
<tr>
<td>CPP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '111'</td>
</tr>
<tr>
<td>DC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_DC</td>
</tr>
<tr>
<td>DVP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp;&amp; op2 == '101'</td>
</tr>
<tr>
<td>IC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_IC</td>
</tr>
<tr>
<td>TLBI</td>
<td>CRn == '1000' &amp;&amp; SysOp(op1,'1000',CRm,op2) == Sys_TLBI</td>
</tr>
</tbody>
</table>

Operation

AArch64.SysInstr(1, sys_op1, sys_crn, sys.crm, sys_op2, t);
### Assembler Symbols

- `<Xt>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- `<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- `<Cn>` Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- `<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- `<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

### Operation

```c
// No architecturally defined instructions here.
AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```
TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

<table>
<thead>
<tr>
<th>b5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>b40</th>
<th>imm14</th>
<th>Rt</th>
</tr>
</thead>
</table>

\[ \text{op} \]

TBNZ <R><t>, #<imm>, <label>

integer \( t = \text{UInt}(\text{Rt}) \);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);

Assembler Symbols

<\text{R}> \quad \text{Is a width specifier, encoded in “b5”:}

\begin{array}{c|c}
\text{b5} & \text{<R>} \\
0 & W \\
1 & X \\
\end{array}

In assembler source code an ‘X’ specifier is always permitted, but a ‘W’ specifier is only permitted when the bit number is less than 32.

<\text{t}> \quad \text{Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.}

<\text{imm}> \quad \text{Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.}

<\text{label}> \quad \text{Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.}

Operation

\begin{align*}
\text{bits(datasize) operand} &= \text{X}[t]; \\
\text{if operand<bit pos> == op then} \\
\quad \text{BranchTo}(\text{PC}[\] + offset, \text{BranchType_DIR}, \text{TRUE});
\end{align*}
TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

<table>
<thead>
<tr>
<th>b5</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>b40</th>
<th>imm14</th>
<th>Rt</th>
</tr>
</thead>
</table>

**Assembler Symbols**

- **<R>** Is a width specifier, encoded in “b5”:
  - b5 <R>
    - 0: W
    - 1: X
  In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

- **<t>** Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

- **<imm>** Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

- **<label>** Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

**Operation**

```plaintext
integer t = UInt(Rt);
integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);

bits(datasize) operand = X[t];
if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
TLBI

TLB Invalidate operation. For more information, see \texttt{op0==0b01, cache maintenance, TLB maintenance, and address translation instructions}.

This is an alias of \texttt{SYS}. This means:

- The encodings in this description are named to match the encodings of \texttt{SYS}.
- The description of \texttt{SYS} gives the operational pseudocode for this instruction.

\begin{verbatim}
1 1 0 1 0 1 0 0 0 1 0 0 0 0 0 1
\end{verbatim}

\texttt{TLBI <tlbi_op>{, <Xt>}}

is equivalent to

\texttt{SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}}

and is the preferred disassembly when \texttt{SysOp(op1,'1000',CRm,op2) == Sys_TLBI}.

**Assembler Symbols**

- \texttt{<op1>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- \texttt{<Cm>} is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- \texttt{<op2>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- \texttt{<tlbi_op>} is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in "op1:CRm:op2":

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;tlbi_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0001</td>
<td>000</td>
<td>VMALLE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>001</td>
<td>VAE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>010</td>
<td>ASIDE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>011</td>
<td>VAAE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>101</td>
<td>VALE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>001</td>
<td>RVAE11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>011</td>
<td>RVAAE11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>101</td>
<td>RVALE11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>111</td>
<td>RVAALE11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>000</td>
<td>VMALLE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>001</td>
<td>VAE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>010</td>
<td>ASIDE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>011</td>
<td>VAAE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>101</td>
<td>VALE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>111</td>
<td>VAALE11S</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>001</td>
<td>RVAE10S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>011</td>
<td>RVAAE10S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>101</td>
<td>RVALE10S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>001</td>
<td>RVAE1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>011</td>
<td>RVAAE1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>101</td>
<td>RVALE1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>111</td>
<td>RVAALE1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>000</td>
<td>VMALLE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>001</td>
<td>VAE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>010</td>
<td>ASIDE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>011</td>
<td>VAAE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>101</td>
<td>VALE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>111</td>
<td>VAALE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>001</td>
<td>IPAS2E11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>010</td>
<td>RIPAS2E11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>101</td>
<td>IPAS2LE11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>110</td>
<td>RIPAS2LE11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>000</td>
<td>ALLE20S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>001</td>
<td>VAE20S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>100</td>
<td>ALLE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>101</td>
<td>VALE20S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>110</td>
<td>VMALLS12E10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>001</td>
<td>RVAE21S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>101</td>
<td>RVALE21S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>000</td>
<td>ALLE21S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>001</td>
<td>VAE21S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>100</td>
<td>ALLE11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>101</td>
<td>VALE21S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>110</td>
<td>VMALLS12E11S</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>000</td>
<td>IPAS2E10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>001</td>
<td>IPAS2E1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>010</td>
<td>RIPAS2E1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>011</td>
<td>RIPAS2E10S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>100</td>
<td>IPAS2LE10S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>101</td>
<td>IPAS2LE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>110</td>
<td>RIPAS2LE11S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0101</td>
<td>001</td>
<td>RVAE20S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0101</td>
<td>101</td>
<td>RVALE20S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0110</td>
<td>001</td>
<td>RVAE2</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0110</td>
<td>101</td>
<td>RVALE2</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>000</td>
<td>ALLE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>001</td>
<td>VAE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>100</td>
<td>ALLE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>101</td>
<td>VALE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>110</td>
<td>VMALLS12E1</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>000</td>
<td>ALLE30S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>001</td>
<td>VAE30S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>101</td>
<td>VALE30S</td>
<td>FEAT_TLBIO</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>001</td>
<td>RVAE31S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>101</td>
<td>RVALE31S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>op1</td>
<td>CRm</td>
<td>op2</td>
<td>&lt;tlbi_op&gt;</td>
<td>Architectural Feature</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>000</td>
<td>ALLE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>001</td>
<td>VAE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>101</td>
<td>VALE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>001</td>
<td>RVAE30S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>101</td>
<td>RVALE30S</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>001</td>
<td>RVAE3</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>101</td>
<td>RVALE3</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>000</td>
<td>ALLE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>001</td>
<td>VAE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>101</td>
<td>VALE3</td>
<td>-</td>
</tr>
</tbody>
</table>

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the “Rt” field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If \textit{FEAT\_TRF} is not implemented, this instruction executes as a \textit{NOP}.

System
\textit{(FEAT\_TRF)}

\begin{verbatim}
    31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
    1  1  0  1  0  1  0  0  0  0  0  1  1  0  0  1  0  0  0  1  0  0  1  0  1  1  1  1  1  1
\end{verbatim}

\textit{TSB CSYNC}

if !\texttt{HaveSelfHostedTrace}() then \texttt{EndOfInstruction}();

\textbf{Operation}

\texttt{TraceSynchronizationBarrier}();

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12\_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result

: Rn AND imm.

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.

```
| sf | 1 | 1 | 1 | 1 | 0 | 0 | 0 | N | immr | imms | Rn | 1 | 1 | 1 | 1 | Rd |
```

**32-bit (sf == 0 && N == 0)**

TST <Wn>, #<imm>

is equivalent to

ANDS WZR, <Wn>, #<imm>

and is always the preferred disassembly.

**64-bit (sf == 1)**

TST <Xn>, #<imm>

is equivalent to

ANDS XZR, <Xn>, #<imm>

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

The description of ANDS (immediate) gives the operational pseudocode for this instruction.
TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

32-bit \( (sf == 0) \)

TST \(<Wn>, <Wm>{, <shift> #<amount>}\)

is equivalent to

ANDS WZR, \(<Wn>, <Wm>{, <shift> #<amount>}\)

and is always the preferred disassembly.

64-bit \( (sf == 1) \)

TST \(<Xn>, <Xm>{, <shift> #<amount>}\)

is equivalent to

ANDS XZR, \(<Xn>, <Xm>{, <shift> #<amount>}\)

and is always the preferred disassembly.

Assembler Symbols

\(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

\(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

\(<shift>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UBFIZ**

Unsigned Bitfield Insert in Zeros copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<lsb>\) of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

![Register Encoding](image)

**32-bit (sf == 0 & N == 0)**

**UBFIZ <Wd>, <Wn>, #<lsb>, #<width>**

is equivalent to

**UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)**

and is the preferred disassembly when \(\text{UInt(imms)} < \text{UInt(immr)}\).

**64-bit (sf == 1 & N == 1)**

**UBFIZ <Xd>, <Xn>, #<lsb>, #<width>**

is equivalent to

**UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)**

and is the preferred disassembly when \(\text{UInt(imms)} < \text{UInt(immr)}\).

**Assembler Symbols**

- \(<Wd>\): Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\): Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\): Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\): Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<lsb>\): For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- \(<width>\): For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UBFM**

Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If `<imms>` is greater than or equal to `<immr>`, this copies a bitfield of `(imms-<immr>+1)` bits starting from bit position `<immr>` in the source register to the least significant bits of the destination register.

If `<imms>` is less than `<immr>`, this copies a bitfield of `(imms+1)` bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0 & N == 0)

UBFM `<Wd>, <Wn>, #<immr>, #<imms>

### 64-bit (sf == 1 & N == 1)

UBFM `<Xd>, <Xn>, #<immr>, #<imms>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
integer R;
bits(datasize) wmask;
bits(datasize) tmask;
if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;
R = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);
```

#### Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.
- `<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

#### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL (immediate)</td>
<td>32-bit</td>
<td><code>imms != '011111' &amp;&amp; imms + 1 == immr</code></td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>64-bit</td>
<td><code>imms != '111111' &amp;&amp; imms + 1 == immr</code></td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>32-bit</td>
<td><code>imms == '011111'</code></td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>64-bit</td>
<td><code>imms == '111111'</code></td>
</tr>
<tr>
<td>Alias</td>
<td>Of variant</td>
<td>Is preferred when</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>UBFIZ</td>
<td>UInt(imm) &lt; UInt(immr)</td>
<td></td>
</tr>
<tr>
<td>UBFX</td>
<td>BFXPreferred(sf, opc&lt;1&gt;, imm, immr)</td>
<td></td>
</tr>
<tr>
<td>UXTB</td>
<td>imms == '000000' &amp;&amp; immr == '000111'</td>
<td></td>
</tr>
<tr>
<td>UXTH</td>
<td>imms == '000000' &amp;&amp; immr == '011111'</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

```markdown
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// combine extension bits and result bits
X[d] = bot AND tmask;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UBFX

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

• The encodings in this description are named to match the encodings of UBFM.
• The description of UBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0)

UBFX <Wd>, <Wn>, <lsb>, <width>

is equivalent to

UBFM <Wd>, <Wn>, <lsb>, (<lsb>+(<width>-1))

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

64-bit (sf == 1 && N == 1)

UBFX <Xd>, <Xn>, <lsb>, <width>

is equivalent to

UBFM <Xd>, <Xn>, <lsb>, (<lsb>+(<width>-1))

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb> For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32.<lsb>.
For the 64-bit variant: is the width of the bitfield, in the range 1 to 64.<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 imm16</td>
</tr>
</tbody>
</table>

UDF #<imm>

// The imm16 field is ignored by hardware.
UNDEFINED;

Assembler Symbols

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field. The PE ignores the value of this constant.

Operation

// No operation.
**UDIV**

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

32-bit (sf == 0)

UDIV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

UDIV <Xd>, <Xn>, <Xm>

```plaintext
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
```

```
dataSize = if sf == '1' then 64 else 32;
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

**Operation**

```plaintext
bits(dataSize) operand1 = X[n];
bits(dataSize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));
X[d] = result<dataSize-1:0>;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMULL**.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UMADDL** <Xd>, <Wn>, <Wm>, <Xa>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- **<Xa>** Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UMULL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d] = result<63:0>;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

UMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register. This instruction is used by the alias UMNEGL.

<table>
<thead>
<tr>
<th>Ra</th>
<th>Rd</th>
<th>Rn</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Xa>`: Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMNEGL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

\[
\begin{align*}
\text{bits}(32) \text{ operand1} &= X[n]; \\
\text{bits}(32) \text{ operand2} &= X[m]; \\
\text{bits}(64) \text{ operand3} &= X[a];
\end{align*}
\]

integer result;

\[
\text{result} = \text{Int}(\text{operand3, TRUE}) - (\text{Int}(\text{operand1, TRUE}) \times \text{Int}(\text{operand2, TRUE}));
\]

\[
X[d] = \text{result<63:0>};
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>(1)(1)(1)(1)(1)</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UMULH <Xd>, <Xn>, <Xm>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

**Assembler Symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d] = result<127:64>;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMULL**

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of **UMADDL**. This means:

- The encodings in this description are named to match the encodings of **UMADDL**.
- The description of **UMADDL** gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**UMULL** `<Xd>, <Wn>, <Wm>`

is equivalent to

**UMADDL** `<Xd>, <Wn>, <Wm>, XZR`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of **UMADDL** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

### 32-bit

UXTB `<Wd>`, `<Wn>`

is equivalent to

**UBFM** `<Wd>`, `<Wn>`, #0, #7

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 | Rn | Rd
sf opc N immr immr
```

**32-bit**

UXTH `<Wd>, <Wn>`

is equivalent to

UBFM `<Wd>, <Wn>, #0, #15`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of UBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1
```

WFE

// Empty.

Operation

```
Hint_WFE(1, WFxType_WFE);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see Wait For Event mechanism and Send event.

As described in Wait For Event mechanism and Send event, the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

WFET <Xt>

if !HaveFeatWFxT() then UNDEFINED;
integer d = UInt(Rd);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

integer localtimeout = UInt(X[d, 64]);

if Halted() & ConstrIfUnpredictableBool(Unpredictable_WFxTDEBUG) then
  EndOfInstruction();

Hint_WFE(localtimeout, WFxType_WFET);
Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

```c
// Empty.

Operation

**Hint_WFI(1, WFxType_WFI);**
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see Wait For Interrupt. As described in Wait For Interrupt, the execution of a WFIT instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

| System
| (FEAT_WFxT) |
|---|---|
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1 | Rd |

**WFIT <Xt>**

if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

**Operation**

integer localtimeout = UInt(X[d, 64]);

if Halted() & ConstrainsUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFI(localtimeout, WFxType_WFIT);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System
(FEAT_FlagM2)

if !HaveFlagFormatExt() then UNDEFINED;

Operation

bit N = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bz Z = PSTATE.Z AND PSTATE.C;
bz C = PSTATE.C OR PSTATE.Z;
bz V = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI. The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses. It has encodings from 2 classes: Integer and System.

Integer
(FEAT_PAuth)

```
0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1
```

XPACD (D == 1)

```
XPACD <Xd>
```

XPACI (D == 0)

```
XPACI <Xd>
```

```java
boolean data = (D == '1');
integer d = UInt(Rd);
if !HavePACExt() then
    UNDEFINED;
```

System
(FEAT_PAuth)

```
0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1
```

XPACLRI

```
integer d = 30;
boolean data = FALSE;
```

Assembler Symbols

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
if HavePACExt() then
    X[d] = Strip(X[d], data);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability. For more information about the recommended use of this instruction, see The YIELD instruction.

Operation

```
// Empty.

Hint_Yield();
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
A64 -- SIMD and Floating-point Instructions (alphabetic order)

**ABS**: Absolute value (vector).

**ADD (vector)**: Add (vector).

**ADDHN, ADDHN2**: Add returning High Narrow.

**ADDP (scalar)**: Add Pair of elements (scalar).

**ADDP (vector)**: Add Pairwise (vector).

**ADDV**: Add across Vector.

**AESD**: AES single round decryption.

**AESE**: AES single round encryption.

**AESIMC**: AES inverse mix columns.

**AESMC**: AES mix columns.

**AND (vector)**: Bitwise AND (vector).

**BCAX**: Bit Clear and XOR.

**BFCVT**: Floating-point convert from single-precision to BFloat16 format (scalar).

**BFCVTN, BFCVTN2**: Floating-point convert from single-precision to BFloat16 format (vector).

**BFDOT (by element)**: BFloat16 floating-point dot product (vector, by element).

**BFDOT (vector)**: BFloat16 floating-point dot product (vector).

**BFMLALB, BFMLALT (by element)**: BFloat16 floating-point widening multiply-add long (by element).

**BFMLALB, BFMLALT (vector)**: BFloat16 floating-point widening multiply-add long (vector).

**BFMMLA**: BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

**BIC (vector, immediate)**: Bitwise bit Clear (vector, immediate).

**BIC (vector, register)**: Bitwise bit Clear (vector, register).

**BIF**: Bitwise Insert if False.

**BIT**: Bitwise Insert if True.

**BSL**: Bitwise Select.

**CLS (vector)**: Count Leading Sign bits (vector).

**CLZ (vector)**: Count Leading Zero bits (vector).

**CMEQ (register)**: Compare bitwise Equal (vector).

**CMEQ (zero)**: Compare bitwise Equal to zero (vector).

**CMGE (register)**: Compare signed Greater than or Equal (vector).

**CMGE (zero)**: Compare signed Greater than or Equal to zero (vector).

**CMGT (register)**: Compare signed Greater than (vector).

**CMGT (zero)**: Compare signed Greater than zero (vector).

**CMHI (register)**: Compare unsigned Higher (vector).

**CMHS (register)**: Compare unsigned Higher or Same (vector).
CMLE (zero): Compare signed Less than or Equal to zero (vector).
CMLT (zero): Compare signed Less than zero (vector).
CMTST: Compare bitwise Test bits nonzero (vector).
CNT: Population Count per byte.
DUP (element): Duplicate vector element to vector or scalar.
DUP (general): Duplicate general-purpose register to vector.
EOR (vector): Bitwise Exclusive OR (vector).
EOR3: Three-way Exclusive OR.
EXT: Extract vector from pair of vectors.
FABD: Floating-point Absolute Difference (vector).
FABS (scalar): Floating-point Absolute value (scalar).
FABS (vector): Floating-point Absolute value (vector).
FACGE: Floating-point Absolute Compare Greater than or Equal (vector).
FACGT: Floating-point Absolute Compare Greater than (vector).
FADD (scalar): Floating-point Add (scalar).
FADD (vector): Floating-point Add (vector).
FADDP (scalar): Floating-point Add Pair of elements (scalar).
FADDP (vector): Floating-point Add Pairwise (vector).
FCADD: Floating-point Complex Add.
FCCMP: Floating-point Conditional quiet Compare (scalar).
FCCMPE: Floating-point Conditional signaling Compare (scalar).
FCMEQ (register): Floating-point Compare Equal (vector).
FCMEQ (zero): Floating-point Compare Equal to zero (vector).
FCMGE (register): Floating-point Compare Greater than or Equal (vector).
FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).
FCMGT (register): Floating-point Compare Greater than (vector).
FCMGT (zero): Floating-point Compare Greater than zero (vector).
FCMLA: Floating-point Complex Multiply Accumulate.
FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).
FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).
FCMLT (zero): Floating-point Compare Less than zero (vector).
FCMP: Floating-point quiet Compare (scalar).
FCMPE: Floating-point signaling Compare (scalar).
FCSEL: Floating-point Conditional Select (scalar).
FCVT: Floating-point Convert precision (scalar).
FCVTAS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).
FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).
FCVTAU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).
FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).
FCVTLL, FCVTLL2: Floating-point Convert to higher precision Long (vector).
FCVTMS (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).
FCVTMS (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).
FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).
FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).
FCVTNL, FCVTNL2: Floating-point Convert to lower precision Narrow (vector).
FCVTPS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).
FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).
FCVTPU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).
FCVTPU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).
FCVTNXN, FCVTNXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).
FCVTZS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
FCVTZS (scalar, integer): Floating-point Convert to Signed integer, rounding toward Zero (scalar).
FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTZU (scalar, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).
FCVTZU (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).
FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
FDIV (scalar): Floating-point Divide (scalar).
FDIV (vector): Floating-point Divide (vector).
FJCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
FMADD: Floating-point fused Multiply-Add (scalar).
FMAX (scalar): Floating-point Maximum (scalar).
FMAX (vector): Floating-point Maximum (vector).
FMAXNM (scalar): Floating-point Maximum Number (scalar).
FMAXNM (vector): Floating-point Maximum Number (vector).
FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).
FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMAXNMV</td>
<td>Floating-point Maximum Number across Vector.</td>
</tr>
<tr>
<td>FMAXP (scalar)</td>
<td>Floating-point Maximum of Pair of elements (scalar).</td>
</tr>
<tr>
<td>FMAXP (vector)</td>
<td>Floating-point Maximum Pairwise (vector).</td>
</tr>
<tr>
<td>FMAXV</td>
<td>Floating-point Maximum across Vector.</td>
</tr>
<tr>
<td>FMIN (scalar)</td>
<td>Floating-point Minimum (scalar).</td>
</tr>
<tr>
<td>FMIN (vector)</td>
<td>Floating-point minimum (vector).</td>
</tr>
<tr>
<td>FMINNM (scalar)</td>
<td>Floating-point Minimum Number (scalar).</td>
</tr>
<tr>
<td>FMINNM (vector)</td>
<td>Floating-point Minimum Number (vector).</td>
</tr>
<tr>
<td>FMINNMP (scalar)</td>
<td>Floating-point Minimum Number of Pair of elements (scalar).</td>
</tr>
<tr>
<td>FMINNMP (vector)</td>
<td>Floating-point Minimum Number Pairwise (vector).</td>
</tr>
<tr>
<td>FMINNMV</td>
<td>Floating-point Minimum Number across Vector.</td>
</tr>
<tr>
<td>FMINP (scalar)</td>
<td>Floating-point Minimum of Pair of elements (scalar).</td>
</tr>
<tr>
<td>FMINP (vector)</td>
<td>Floating-point Minimum Pairwise (vector).</td>
</tr>
<tr>
<td>FMINV</td>
<td>Floating-point Minimum across Vector.</td>
</tr>
<tr>
<td>FMLA (by element)</td>
<td>Floating-point fused Multiply-Add to accumulator (by element).</td>
</tr>
<tr>
<td>FMLA (vector)</td>
<td>Floating-point fused Multiply-Add to accumulator (vector).</td>
</tr>
<tr>
<td>FMLAL, FMLAL2 (by element)</td>
<td>Floating-point fused Multiply-Add Long to accumulator (by element).</td>
</tr>
<tr>
<td>FMLAL, FMLAL2 (vector)</td>
<td>Floating-point fused Multiply-Add Long to accumulator (vector).</td>
</tr>
<tr>
<td>FMLS (by element)</td>
<td>Floating-point fused Multiply-Subtract from accumulator (by element).</td>
</tr>
<tr>
<td>FMLS (vector)</td>
<td>Floating-point fused Multiply-Subtract from accumulator (vector).</td>
</tr>
<tr>
<td>FMLSL, FMLSL2 (by element)</td>
<td>Floating-point fused Multiply-Subtract Long from accumulator (by element).</td>
</tr>
<tr>
<td>FMLSL, FMLSL2 (vector)</td>
<td>Floating-point fused Multiply-Subtract Long from accumulator (vector).</td>
</tr>
<tr>
<td>FMOV (general)</td>
<td>Floating-point Move to or from general-purpose register without conversion.</td>
</tr>
<tr>
<td>FMOV (register)</td>
<td>Floating-point Move register without conversion.</td>
</tr>
<tr>
<td>FMOV (scalar, immediate)</td>
<td>Floating-point move immediate (scalar).</td>
</tr>
<tr>
<td>FMOV (vector, immediate)</td>
<td>Floating-point move immediate (vector).</td>
</tr>
<tr>
<td>FMSUB</td>
<td>Floating-point Fused Multiply-Subtract (scalar).</td>
</tr>
<tr>
<td>FMUL (by element)</td>
<td>Floating-point Multiply (by element).</td>
</tr>
<tr>
<td>FMUL (scalar)</td>
<td>Floating-point Multiply (scalar).</td>
</tr>
<tr>
<td>FMUL (vector)</td>
<td>Floating-point Multiply (vector).</td>
</tr>
<tr>
<td>FMULX</td>
<td>Floating-point Multiply extended.</td>
</tr>
<tr>
<td>FMULX (by element)</td>
<td>Floating-point Multiply extended (by element).</td>
</tr>
<tr>
<td>FNNEG (scalar)</td>
<td>Floating-point Negate (scalar).</td>
</tr>
<tr>
<td>FNNEG (vector)</td>
<td>Floating-point Negate (vector).</td>
</tr>
<tr>
<td>FNMADD</td>
<td>Floating-point Negated fused Multiply-Add (scalar).</td>
</tr>
<tr>
<td>FNMSUB</td>
<td>Floating-point Negated fused Multiply-Subtract (scalar).</td>
</tr>
</tbody>
</table>
FNMUL (scalar): Floating-point Multiply-Negate (scalar).
FRECPE: Floating-point Reciprocal Estimate.
FRECPS: Floating-point Reciprocal Step.
FRECPX: Floating-point Reciprocal exponent (scalar).
FRINT32X (scalar): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).
FRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).
FRINT32Z (scalar): Floating-point Round to 32-bit Integer toward Zero (scalar).
FRINT32Z (vector): Floating-point Round to 32-bit Integer toward Zero (vector).
FRINT64X (scalar): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).
FRINT64X (vector): Floating-point Round to 64-bit Integer, using current rounding mode (vector).
FRINT64Z (scalar): Floating-point Round to 64-bit Integer toward Zero (scalar).
FRINT64Z (vector): Floating-point Round to 64-bit Integer toward Zero (vector).
FRINTA (scalar): Floating-point Round to Integral, to nearest with ties to Away (scalar).
FRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).
FRINTI (scalar): Floating-point Round to Integral, using current rounding mode (scalar).
FRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).
FRINTM (scalar): Floating-point Round to Integral, toward Minus infinity (scalar).
FRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).
FRINTN (scalar): Floating-point Round to Integral, to nearest with ties to even (scalar).
FRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).
FRINTP (scalar): Floating-point Round to Integral, toward Plus infinity (scalar).
FRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).
FRINTX (scalar): Floating-point Round to Integral exact, using current rounding mode (scalar).
FRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).
FRINTZ (scalar): Floating-point Round to Integral, toward Zero (scalar).
FRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).
FRSORTE: Floating-point Reciprocal Square Root Estimate.
FRSQRTS: Floating-point Reciprocal Square Root Step.
FSORT (scalar): Floating-point Square Root (scalar).
FSORT (vector): Floating-point Square Root (vector).
FSUB (scalar): Floating-point Subtract (scalar).
FSUB (vector): Floating-point Subtract (vector).
INS (element): Insert vector element from another vector element.
INS (general): Insert vector element from general-purpose register.
LDI (multiple structures): Load multiple single-element structures to one, two, three, or four registers.
LDI (single structure): Load one single-element structure to one lane of one register.
LD1R: Load one single-element structure and Replicate to all lanes (of one register).
LD2 (multiple structures): Load multiple 2-element structures to two registers.
LD2 (single structure): Load single 2-element structure to one lane of two registers.
LD2R: Load single 2-element structure and Replicate to all lanes of two registers.
LD3 (multiple structures): Load multiple 3-element structures to three registers.
LD3 (single structure): Load single 3-element structure to one lane of three registers.
LD3R: Load single 3-element structure and Replicate to all lanes of three registers.
LD4 (multiple structures): Load multiple 4-element structures to four registers.
LD4 (single structure): Load single 4-element structure to one lane of four registers.
LD4R: Load single 4-element structure and Replicate to all lanes of four registers.
LDNP (SIMD&FP): Load Pair of SIMD&FP registers, with Non-temporal hint.
LDP (SIMD&FP): Load Pair of SIMD&FP registers.
LDR (immediate, SIMD&FP): Load SIMD&FP Register (immediate offset).
LDR (literal, SIMD&FP): Load SIMD&FP Register (PC-relative literal).
LDR (register, SIMD&FP): Load SIMD&FP Register (register offset).
LDUR (SIMD&FP): Load SIMD&FP Register (unscaled offset).
MLA (by element): Multiply-Add to accumulator (vector, by element).
MLA (vector): Multiply-Add to accumulator (vector).
MLS (by element): Multiply-Subtract from accumulator (vector, by element).
MLS (vector): Multiply-Subtract from accumulator (vector).
MOV (element): Move vector element to another vector element: an alias of INS (element).
MOV (from general): Move general-purpose register to a vector element: an alias of INS (general).
MOV (scalar): Move vector element to scalar: an alias of DUP (element).
MOV (to general): Move vector element to general-purpose register: an alias of UMOV.
MOVI: Move Immediate (vector).
MUL (by element): Multiply (vector, by element).
MUL (vector): Multiply (vector).
MVN: Bitwise NOT (vector): an alias of NOT.
MVNI: Move inverted Immediate (vector).
NEG (vector): Negate (vector).
NOT: Bitwise NOT (vector).
ORN (vector): Bitwise inclusive OR NOT (vector).
ORR (vector, immediate): Bitwise inclusive OR (vector, immediate).
ORR (vector, register): Bitwise inclusive OR (vector, register).
PMUL: Polynomial Multiply.
PMULL, PMULL2: Polynomial Multiply Long.
RADDHN, RADDHN2: Rounding Add returning High Narrow.
RAX1: Rotate and Exclusive OR.
RBIT (vector): Reverse Bit order (vector).
REV16 (vector): Reverse elements in 16-bit halfwords (vector).
REV32 (vector): Reverse elements in 32-bit words (vector).
REV64: Reverse elements in 64-bit doublewords (vector).
RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).
RSUBHN, RSUBHN2: Rounding Subtract returning High Narrow.
SABA: Signed Absolute difference and Accumulate.
SABAL, SABAL2: Signed Absolute difference and Accumulate Long.
SABD: Signed Absolute Difference.
SABDL, SABDL2: Signed Absolute Difference Long.
SADALP: Signed Add and Accumulate Long Pairwise.
SADDL, SADDL2: Signed Add Long (vector).
SADDP: Signed Add Long Pairwise.
SADDLV: Signed Add Long across Vector.
SADDW, SADDW2: Signed Add Wide.
SCVTF (scalar, fixed-point): Signed fixed-point Convert to Floating-point (scalar).
SCVTF (scalar, integer): Signed integer Convert to Floating-point (scalar).
SCVTF (vector, fixed-point): Signed fixed-point Convert to Floating-point (vector).
SCVTF (vector, integer): Signed integer Convert to Floating-point (vector).
SDOT (by element): Dot Product signed arithmetic (vector, by element).
SDOT (vector): Dot Product signed arithmetic (vector).
SHA1C: SHA1 hash update (choose).
SHA1H: SHA1 fixed rotate.
SHA1M: SHA1 hash update (majority).
SHA1P: SHA1 hash update (parity).
SHA1SU0: SHA1 schedule update 0.
SHA1SU1: SHA1 schedule update 1.
SHA256H: SHA256 hash update (part 1).
SHA256H2: SHA256 hash update (part 2).
SHA256SU0: SHA256 schedule update 0.
SHA256SU1: SHA256 schedule update 1.
SHA512H: SHA512 Hash update part 1.
SHA512H2: SHA512 Hash update part 2.
SHA512SU0: SHA512 Schedule Update 0.
SHA512SU1: SHA512 Schedule Update 1.
SHADD: Signed Halving Add.
SHL: Shift Left (immediate).
SHLL, SHLL2: Shift Left Long (by element size).
SHRN, SHRN2: Shift Right Narrow (immediate).
SHSUB: Signed Halving Subtract.
SII: Shift Left and Insert (immediate).
SM3PARTW1: SM3PARTW1.
SM3PARTW2: SM3PARTW2.
SM3SS1: SM3SS1.
SM3TT1A: SM3TT1A.
SM3TT1B: SM3TT1B.
SM3TT2A: SM3TT2A.
SM3TT2B: SM3TT2B.
SM4E: SM4 Encode.
SM4EKEY: SM4 Key.
SMAX: Signed Maximum (vector).
SMAXP: Signed Maximum Pairwise.
SMAXV: Signed Maximum across Vector.
SMIN: Signed Minimum (vector).
SMINP: Signed Minimum Pairwise.
SMINV: Signed Minimum across Vector.
SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).
SMMLA (vector): Signed 8-bit integer matrix multiply-accumulate (vector).
SMOV: Signed Move vector element to general-purpose register.
SQABS: Signed saturating Absolute value.
SQADD: Signed saturating Add.
SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).

SQDMLUH (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMLUH (vector): Signed saturating Doubling Multiply returning High half.

SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).


SQNEG: Signed saturating Negate.

SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

SQRDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half.

SQRSHL: Signed saturating Rounding Shift Left (register).

SQRSHRN, SQRSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).

SQRSHRUN, SQRSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

SQRSHL (immediate): Signed saturating Shift Left (immediate).

SQRSHL (register): Signed saturating Shift Left (register).

SQRSHLU: Signed saturating Shift Left Unsigned (immediate).

SQRSHRN, SQRSHRN2: Signed saturating Shift Right Narrow (immediate).

SQRSHRUN, SQRSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).

SQRSUB: Signed saturating Subtract.

SQXTN, SQXTN2: Signed saturating extract Narrow.

SOXTUN, SOXTUN2: Signed saturating extract Unsigned Narrow.

SRHADD: Signed Rounding Halving Add.

SRI: Shift Right and Insert (immediate).

SRSRL: Signed Rounding Shift Left (register).

SRSRH: Signed Rounding Shift Right (immediate).

SRSRA: Signed Rounding Shift Right and Accumulate (immediate).

SSHLL, SSHLL2: Signed Shift Left Long (immediate).

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

ST1 (multiple structures): Store multiple single-element structures from one, two, three, or four registers.

ST1 (single structure): Store a single-element structure from one lane of one register.
**ST2 (multiple structures)**: Store multiple 2-element structures from two registers.

**ST2 (single structure)**: Store single 2-element structure from one lane of two registers.

**ST3 (multiple structures)**: Store multiple 3-element structures from three registers.

**ST3 (single structure)**: Store single 3-element structure from one lane of three registers.

**ST4 (multiple structures)**: Store multiple 4-element structures from four registers.

**ST4 (single structure)**: Store single 4-element structure from one lane of four registers.

**STNP (SIMD&FP)**: Store Pair of SIMD&FP registers, with Non-temporal hint.

**STP (SIMD&FP)**: Store Pair of SIMD&FP registers.

**STR (immediate, SIMD&FP)**: Store SIMD&FP register (immediate offset).

**STR (register, SIMD&FP)**: Store SIMD&FP register (register offset).

**STUR (SIMD&FP)**: Store SIMD&FP register (unscaled offset).

**SUB (vector)**: Subtract (vector).

**SUBHN, SUBHN2**: Subtract returning High Narrow.

**SUDOT (by element)**: Dot product with signed and unsigned integers (vector, by element).

**SUQADD**: Signed saturating Accumulate of Unsigned value.

**SXTL, SXTL2**: Signed extend Long: an alias of SSHLL, SSHLL2.

**TBL**: Table vector Lookup.

**TBX**: Table vector lookup extension.

**TRN1**: Transpose vectors (primary).

**TRN2**: Transpose vectors (secondary).

**UABA**: Unsigned Absolute difference and Accumulate.

**UABAL, UABAL2**: Unsigned Absolute difference and Accumulate Long.

**UABD**: Unsigned Absolute Difference (vector).

**UABDL, UABDL2**: Unsigned Absolute Difference Long.

**UADALP**: Unsigned Add and Accumulate Long Pairwise.

**UADDL, UADDL2**: Unsigned Add Long (vector).

**UADLP**: Unsigned Add Long Pairwise.

**UADLV**: Unsigned sum Long across Vector.

**UADDW, UADDW2**: Unsigned Add Wide.

**UCVT (scalar, fixed-point)**: Unsigned fixed-point Convert to Floating-point (scalar).

**UCVT (scalar, integer)**: Unsigned integer Convert to Floating-point (scalar).

**UCVT (vector, fixed-point)**: Unsigned fixed-point Convert to Floating-point (vector).

**UCVT (vector, integer)**: Unsigned integer Convert to Floating-point (vector).

**UDOT (by element)**: Dot Product unsigned arithmetic (vector, by element).

**UDOT (vector)**: Dot Product unsigned arithmetic (vector).

**UHADD**: Unsigned Halving Add.
**UHSUB**: Unsigned Halving Subtract.

**UMAX**: Unsigned Maximum (vector).

**UMAXP**: Unsigned Maximum Pairwise.

**UMAXV**: Unsigned Maximum across Vector.

**UMIN**: Unsigned Minimum (vector).

**UMINP**: Unsigned Minimum Pairwise.

**UMINV**: Unsigned Minimum across Vector.

**UMLAL, UMLAL2 (by element)**: Unsigned Multiply-Add Long (vector, by element).

**UMLAL, UMLAL2 (vector)**: Unsigned Multiply-Add Long (vector).

**UMLSL, UMLSL2 (by element)**: Unsigned Multiply-Subtract Long (vector, by element).

**UMLSL, UMLSL2 (vector)**: Unsigned Multiply-Subtract Long (vector).

**UMMLA (vector)**: Unsigned 8-bit integer matrix multiply-accumulate (vector).

**UMOV**: Unsigned Move vector element to general-purpose register.

**UMULL, UMULL2 (by element)**: Unsigned Multiply Long (vector, by element).

**UMULL, UMULL2 (vector)**: Unsigned Multiply long (vector).

**UQADD**: Unsigned saturating Add.

**UQRSHL**: Unsigned saturating Rounding Shift Left (register).

**UQRSHRN, UQRSHRN2**: Unsigned saturating Rounded Shift Right Narrow (immediate).

**UQSHL (immediate)**: Unsigned saturating Shift Left (immediate).

**UQSHL (register)**: Unsigned saturating Shift Left (register).

**UQSHRN, UQSHRN2**: Unsigned saturating Shift Right Narrow (immediate).

**UQSUB**: Unsigned saturating Subtract.

**UQXTN, UQXTN2**: Unsigned saturating extract Narrow.

**URECPE**: Unsigned Reciprocal Estimate.

**URHADD**: Unsigned Rounding Halving Add.

**URSHL**: Unsigned Rounding Shift Left (register).

**URSHR**: Unsigned Rounding Shift Right (immediate).

**URSORTE**: Unsigned Reciprocal Square Root Estimate.

**URSRA**: Unsigned Rounding Shift Right and Accumulate (immediate).

**USDOT (by element)**: Dot Product with unsigned and signed integers (vector, by element).

**USDOT (vector)**: Dot Product with unsigned and signed integers (vector).

**USHL**: Unsigned Shift Left (register).

**USHLL, USHLL2**: Unsigned Shift Left Long (immediate).

**USHR**: Unsigned Shift Right (immediate).

**USMMLA (vector)**: Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).

**USQADD**: Unsigned saturating Accumulate of Signed value.
**USRA**: Unsigned Shift Right and Accumulate (immediate).

**USUBL, USUBL2**: Unsigned Subtract Long.

**USUBW, USUBW2**: Unsigned Subtract Wide.

**UXTL, UXTL2**: Unsigned extend Long: an alias of USHLL, USHLL2.

**UZP1**: Unzip vectors (primary).

**UZP2**: Unzip vectors (secondary).

**XAR**: Exclusive OR and Rotate.

**XTN, XTN2**: Extract Narrow.

**ZIP1**: Zip vectors (primary).

**ZIP2**: Zip vectors (secondary).
**ABS**

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: Scalar and Vector.

**Scalar**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

ABS `<V>`<d>, `<V>`<n>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

**Vector**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

ABS `<Vd>.<T>`, `<Vn>.<T>`

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

---

ABS
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> \text{ \ is the name of the SIMD&FP source register, encoded in the "Rn" field.}

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
in integer element;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
0 1 0 1 1 1 1 0 | size 1 | Rm 1 0 0 0 0 1 | Rn 1 0 0 0 0 1 | Rd
```

```
ADD <V><d>, <V><n>, <V><m>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');

Vector

```
0 1 0 1 1 1 1 0 | size 1 | Rm 1 0 0 0 0 1 | Rn 1 0 0 0 0 1 | Rd
```

```
ADD <Vd><T>, <Vn><T>, <Vm><T>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');

Assembler Symbols

`<V>` Is a width specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

`<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

`<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

`<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if sub_op then
    Elem[result, e, esize] = element1 - element2;
  else
    Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see \textit{RADDHN}.

The \textit{ADDHN} instruction writes the vector to the lower half of the destination register and clears the upper half, while the \textit{ADDHN2} instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the \textit{CPACR_EL1}, \textit{CPTR_EL2}, and \textit{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
ADDHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
\end{verbatim}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean round = (U == '1');
\end{verbatim}

\textbf{Assembler Symbols}

2 \hspace{1cm} Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

\begin{verbatim}
Q 0 1
\end{verbatim}

\begin{verbatim}
[absent] [present]
\end{verbatim}

\textbf{<Vd>} \hspace{1cm} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textbf{<Tb>} \hspace{1cm} Is an arrangement specifier, encoded in "size:Q":

\begin{verbatim}
size Q <Tb>
00 0  8B
00 1  16B
01 0  4H
01 1  8H
10 0  2S
10 1  4S
11 x  RESERVED
\end{verbatim}

\textbf{<Vn>} \hspace{1cm} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\textbf{<Ta>} \hspace{1cm} Is an arrangement specifier, encoded in "size":

\begin{verbatim}
size <Ta>
00  8H
01  4S
10  2D
11  RESERVED
\end{verbatim}

\textbf{<Vm>} \hspace{1cm} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembly Symbols

<ADDP> <V>d>, <Vn>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize * 2;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>Q</th>
<th>Rm</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << Uint(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 0  | 0  | 0  | 1  | 1  | 1  | 1  |   |   |   |   |   |   |   |   | Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

**Assembler Symbols**

- `<V>`: Is the destination width specifier, encoded in "size":
  - `size`<V>
    - 00: B
    - 01: H
    - 10: S
    - 11: RESERVED

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<T>`: Is an arrangement specifier, encoded in "size:Q":
  - `size` Q <T>
    - 00 0: 8B
    - 00 1: 16B
    - 01 0: 4H
    - 01 1: 8H
    - 10 0: RESERVED
    - 10 1: 4S
    - 11 x: RESERVED

**Operation**

```assembly
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESD

AES single round decryption.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 6 |

AESD <Vd>.16B, <Vn>.16B

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESE

AES single round encryption.

Integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result = AESSubBytes(AESShiftRows(result));
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
AES inverse mix columns.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  0 |  1 |  0 |  1 |  1 |  0 |  0 |  0 |  1 |  0 |  1 |  0 |  0 |  0 |  1 |  1 |  1 |  1 |  0 |   |   |   |   |   |   |   |   | D |

AESIMC <Vd>.16B, <Vn>.16B

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
result = AESInvMixColumns(operand);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESMC

AES mix columns.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | Rn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

D

AESMC \texttt{<Vd>}, \texttt{<Vn>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
\end{verbatim}

Assembler Symbols

\begin{itemize}
\item \texttt{<Vd>} is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\item \texttt{<Vn>} is the name of the SIMD&FP source register, encoded in the "Rn" field.
\end{itemize}

Operation

\begin{verbatim}
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = \texttt{V}[n];
bits(128) result;
result = AESMixColumns(operand);
\texttt{V}[d] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:

\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | Rn | 0  | Rd |

size

AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
result = operand1 AND operand2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register. This instruction is implemented only when FEAT_SHA3 is implemented.

Advanced SIMD (FEAT_SHA3)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  |   | Ra |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Rm  
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Rn  
| 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Rd  
```

BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Va>** Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR (Vm AND NOT(Va));
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFCVT

Floating-point convert from single-precision to BFloat16 format (scalar) converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

ID AA64ISARI_EL1.BF16 indicates whether this instruction is supported.

Single-precision to BFloat16

(Feat_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0 0 1 1 0 1 0 0 0</td>
<td>Rd</td>
</tr>
</tbody>
</table>

BFCVT <Hd>, <Sn>

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(32) operand = V[n];
FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

Elem[result, 0, 16] = FPConvertBF(operand, fpcr);

V[d] = result;
BFCVTN, BFCVTN2

Floating-point convert from single-precision to BFloat16 format (vector) reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The BFCVTN instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero, while the BFCVTN2 instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

Vector single-precision to BFloat16
(FEAT_BF16)

<table>
<thead>
<tr>
<th></th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>Q</td>
<td>2</td>
</tr>
</tbody>
</table>

BFCVTN{2} <Vd>, <Ta>, <Vn>.4S

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer part = UInt(Q);
integer elements = 64 DIV 16;

Assembler Symbols

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

|        |        |
| 0      | [absent] |
| 1      | [present] |

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

|        |        |
| 0      | 4H     |
| 1      | 8H     |

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) operand = V[n];
bits(64) result;
for e = 0 to elements-1
  Elem[result, e, 16] = FPConvertBF(Elem[operand, e, 32], FPCR[]);
Vpart[d, part] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of BFloat16 elements.

Irrespective of the control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Generates only the default NaN, as if FPCR.DN is 1.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.

The BFloat16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

**Vector (FEAT_BF16)**

```
     31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
O  Q 0 0 1 1 1 1 0 1 L M Rm 1 1 1 1 H 0 Rn  Rd
```

BFDOT `<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[index]>`

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;Ta&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- `<index>` Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];

    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFDOT (vector)

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of BFloat16 elements. Irrespective of the control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if FPCR.DN is 1.

Vector (FEAT_BF16)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Q   | 1   | 1   | 1   | 1   | 0   | 0   | 1   | 0   | Rm  | 1   | 1   | 1   | 1   | 1   | Rn  | Rd  |
```

BFDOT <Vd>..<Ta>, <Vn>..<Tb>, <Vm>..<Tb>

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` is an arrangement specifier, encoded in “Q”:

```
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>
```

- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` is an arrangement specifier, encoded in “Q”:

```
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
```

- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*e+1, 16];
    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
V[d] = result;
BFMLALB, BFMLALT (by element)

BFLOAT16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from BFLOAT16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the first source vector.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(VEAT_BF16)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BFMLALB<bt> <Vd>.4S<bt>, <VN>.8H, <Vm>.H<index>

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

Assembler Symbols

&lt;bt&gt; Is the bottom or top element specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;bt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

&lt;Vd&gt; Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

&lt;Vn&gt; Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

&lt;Vm&gt; Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

&lt;index&gt; Is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) result;
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(32) element2 = Elem[operand2, index, 16]:Zeros(16);
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(addend, element1, element2, FPCR[]);
V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFMLALB, BFMLALT (vector)

BFLOAT16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from BFloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. **ID_AA64ISAR1_EL1** BF16 indicates whether this instruction is supported.

### Vector (FEAT_BF16)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BFMLAL** <bt> <Vd>.4S, <Vn>.8H, <Vm>.8H

if !HaveBF16Ext() then UNDEFINED;
integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
integer elements = 128 DIV 32;
sel = UInt(Q);

### Assembler Symbols

- <bt> Is the bottom or top element specifier, encoded in “Q”:
  - 0 B
  - 1 T

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(128) result;

for e = 0 to elements-1
  bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
  bits(32) element2 = Elem[operand2, 2*e+sel, 16]:Zeros(16);
  bits(32) addend = Elem[operand3, e, 32];
  Elem[result, e, 32] = BFMulAdd(addend, element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFMMLA

BFLOAT16 floating-point matrix multiply-accumulate into 2x2 matrix.
Irrespective of the control bits in the FPCR, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFLOAT16 elements while multiplying the 2x4 matrix of BFLOAT16 values in the first source vector with the 4x2 matrix of BFLOAT16 values in the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if FPCR.DN is 1.

Note

Arm expects that the BFMMLA instruction will deliver a peak BFLOAT16 multiply throughput that is at least as high as can be achieved using two BFDOT instructions, with a goal that it should have significantly higher throughput.

Vector
(FEAT_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

BFMMLA <Vd>.4S, <Vn>.8H, <Vm>.8H

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);

Assembler Symbols

- <Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) op1 = V[n];
bits(128) op2 = V[m];
bits(128) acc = V[d];
V[d] = BFMatMulAdd(acc, op1, op2);
BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | a | b | c | x | x | x | 1 | 0 | 1 | d | e | f | g | h | Rd
```

**16-bit (cmode == 10x1)**

BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}

**32-bit (cmode == 0xx1)**

BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;

case cmode:op of
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11111' // FMOV Dn,#imm is in main FP instruction set
  if Q == '0' then UNDEFINED;
  operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP register, encoded in the “Rd” field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:
defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);
V[rd] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | 1  | Rn | 0  | 0  | 0  | 1  | 1  | Rd |

size

BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 AND operand2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 1 1 1 0 1 1 1 | Rm | 0 0 0 1 1 1 | Rn | Rd

op2
```

BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “Q”:
  - Q | <T>
  - 0 | 8B
  - 1 | 16B
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];
operand1 = V[d];
operand3 = NOT(V[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**BIT**

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

\[
\begin{array}{cccccccccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \text{Rm} & 0 & 0 & 0 & 1 & 1 & 1 & \text{Rn} & \text{Rd} \\
\end{array}
\]

\[\text{opc2} \]

**BIT** \(<Vd>..<T>, <Vn>..<T>, <Vm>..<T>\)

integer \(d = \text{UInt}(\text{Rd})\);  
integer \(n = \text{UInt}(\text{Rn})\);  
integer \(m = \text{UInt}(\text{Rm})\);  
integer datasize = if \(Q == '1'\) then 128 else 64;

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
\text{CheckFPAdvSIMDEnabled64()};
\text{bits(datasize) operand1;}
\text{bits(datasize) operand3;}
\text{bits(datasize) operand4 = V[n];}
\text{operand1 = V[d];}
\text{operand3 = V[m];}
\text{V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);} 
```

**Operational Information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

---

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 0 | 1 | 1 | 0 | 0 | 1 | 1 | Rm | 0 | 0 | 0 | 1 | 1 | 1 | Rn | Rd

opc2

BSL \texttt{<Vd>.<T>}, \texttt{<Vn>.<T>}, \texttt{<Vm>.<T>}
\end{verbatim}

\texttt{integer d = UInt(Rd);} \\
\texttt{integer n = UInt(Rn);} \\
\texttt{integer m = UInt(Rm);} \\
\texttt{integer datasize = if Q == '1' then 128 else 64;}

\textbf{Assembler Symbols}

\begin{verbatim}
\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field. \\
\texttt{<T>} Is an arrangement specifier, encoded in “Q”:
\end{verbatim}

\begin{verbatim}

\begin{array}{c|c}
Q & <T> \\
\hline
0 & 8B \\
1 & 16B \\
\end{array}

\texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field. \\
\texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\end{verbatim}

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64(); \\
bits(datasize) operand1; \\
bits(datasize) operand3; \\
bits(datasize) operand4 = V[n]; \\
operand1 = V[m]; \\
operand3 = V[d]; \\
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3); \\
\end{verbatim}

\textbf{Operational information}

If \texttt{PSTATE.DIT} is 1:

\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
**CLS (vector)**

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | size| 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

**<Vd>**, **<Vn>**

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);

if size == '11' then \texttt{UNDEFINED};
integer esize = 8 << \text{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then \texttt{CountOp_CLZ} else \texttt{CountOp_CLS};

**Assembler Symbols**

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVEd</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\[
\text{CheckFPAdvSIMDEnabled64}();
\text{bits}(\text{datasize}) \text{ operand} = V[n];
\text{bits}(\text{datasize}) \text{ result};
\]

integer count;
for e = 0 to elements-1
    if countop == \texttt{CountOp_CLS} then
        count = \texttt{CountLeadingSignBits} (\texttt{Elem}[operand, e, esize]);
    else
        count = \texttt{CountLeadingZeroBits} (\texttt{Elem}[operand, e, esize]);
        \texttt{Elem}[result, e, esize] = count<esize-1:0>;
\]

\[
V[d] = \text{result};
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its registers.

The values of the NZCV flags.
CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | size| 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

CLZ <Vd>..<T>, <Vn>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    E[lm result, e, esize] = count<esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | size| 1  | Rm | 1  | 0  | 0  | 0  | 1  | 1  | Rn | Rd |

U

CMEQ <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 0  | size| 1  | Rm | 1  | 0  | 0  | 0  | 1  | 1  | Rn | Rd |

U

CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier; encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the \textit{CPACR\_EL1, CPTR\_EL2,} and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CMEQ** \(<V><d>, <V><n>\), \#0

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if size \(!= '11'\) then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;
case op:U of
  when '00' comparison = \text{CompareOp\_GT};
  when '01' comparison = \text{CompareOp\_GE};
  when '10' comparison = \text{CompareOp\_EQ};
  when '11' comparison = \text{CompareOp\_LE};

### Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CMEQ** \(<Vd>..<T>, <Vn>..<T>\), \#0

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**CompareOp** comparison;
case op:U of
  when '00' comparison = \text{CompareOp\_GT};
  when '01' comparison = \text{CompareOp\_GE};
  when '10' comparison = \text{CompareOp\_EQ};
  when '11' comparison = \text{CompareOp\_LE};

**Assembler Symbols**

\(<V>\) Is a width specifier, encoded in "size":
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 1 1 1 0   | size            | Rm              | 0 0 1 1 1 1     | Rn              | Rd              |
| U               | eq              |                 |                 |                 |                 |
```

CMGE <V<d>, <V<n>, <V<m>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembling

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 Q 0 1 1 1 0   | size            | Rm              | 0 0 1 1 1 1     | Rn              | Rd              |
| U               | eq              |                 |                 |                 |                 |
```

CMGE <V<d>.<T>, <V<n>.<T>, <V<m>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembling Symbols

```
<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
<th>8x</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
<td></td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | size | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | Rn | Rd |

**CMGE <V><d>, <V><n>, #0**

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | size | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | Rn | Rd |

**CMGE <Vd>..<T>, <Vn>..<T>, #0**

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**CompareOp** comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

### Assembler Symbols

<V> Is a width specifier, encoded in “size”:
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 1 1 1 0 size 1 Rm 0 0 1 1 0 0 1 1 0 1 0 Rd
   U               eq
```

CMGT `<V><d>`, `<V><n>`, `<V><m>`

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 Q 0 0 1 1 1 0 size 1 Rm 0 0 1 1 0 0 1 1 0 1 1 Rd
   U               eq
```

CMGT `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<T>`

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Assembler Symbols

- `<V>` is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0               | 1               | 0               | 1               | 1               | 1               | 1               | 0               | 0               | 0               | 0               | 0               | 1               | 0               | 0               | 1               | 0               | Rn               | Rd               |
```

```
CMGT <V><d>, <V><n>, #0
```

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

**Vector**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0               | 1               | 0               | 1               | 1               | 1               | 1               | 0               | 0               | 0               | 0               | 0               | 1               | 0               | 0               | 1               | 0               | Rn               | Rd               |
```

```
CMGT <Vd>.<T>, <Vn>.<T>, #0
```

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**CompareOp** comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

**Assembler Symbols**

```
<V>\ Is a width specifier, encoded in “size”:
```
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n>
Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>
Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

<V> is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> is the number of the SIMD&FP destination register, in the "Rd" field.

<n> is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

**Scalar**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size Rm 0 0 1 1 1 1 Rn Rd
U eq
```

CMHS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

**Vector**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size Rm 0 0 1 1 1 1 Rn Rd
U eq
```

CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

**Assembler Symbols**

```
<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 1 0 | size | 1 0 0 0 0 | 0 1 0 0 | 1 1 0 | Rn | Rd |

U

<table>
<thead>
<tr>
<th>op</th>
</tr>
</thead>
</table>

CMLE <V><d>, <V><n>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 0 1 1 1 0 | size | 1 0 0 0 0 | 0 1 0 0 | 1 1 0 | Rn | Rd |

U

| op |

CMLE <Vd>.<T>, <Vn>.<T>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

**Assembler Symbols**

<V> Is a width specifier, encoded in “size”:
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
0 1 0 1 1 1 1 0 size 1 0 0 0 0 0 1 0 1 0 1 0 Rd
```

CMLT <V>d>, <V>n>, #0

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

### Vector

```
0 Q 0 0 1 1 1 1 size 1 0 0 0 0 0 1 0 1 0 1 0 Rd
```

CMLT <Vd>.<T>, <Vn>.<T>, #0

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- `<T>` Is an arrangement specifier, encoded in “size:Q”:
The name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**CMTST**

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
0 1 0 1 1 1 0 | size | 1 | Rm | 1 0 0 0 1 1 | Rn | Rd
```

CMTST <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

**Vector**

```
0 1 0 1 1 1 0 | size | 1 | Rm | 1 0 0 0 1 1 | Rn | Rd
```

CMTST <Vd><T>, <Vn><T>, <Vm><T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

**Assembler Symbols**

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CNT

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------|----------|----------|
| 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0     |
| size | Rd | Rn |
```

CNT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias \texttt{MOV (scalar)}.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}.

\textbf{Scalar}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0 0 0 0 \Texttt{imm5} 0 0 0 0 0 1 \Texttt{Rn} \Texttt{Rd}
\end{verbatim}

\texttt{DUP <Vd>, <Vn>.<T>[<index>]}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if \texttt{imm5<4>} == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
\end{verbatim}

\textbf{Vector}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 1 1 1 0 0 0 0 \Texttt{imm5} 0 0 0 0 0 1 \Texttt{Rn} \Texttt{Rd}
\end{verbatim}

\texttt{DUP <Vd>.<T>, <Vn>.<Ts>[<index>]}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if \texttt{imm5<4>} == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
\end{verbatim}

\textbf{Assembler Symbols}

\texttt{<T>}

For the scalar variant: is the element width specifier, encoded in “imm5”:

\begin{verbatim}
\begin{tabular}{|c|c|}
\hline
\texttt{imm5} & \texttt{<T>} \\
\hline
0000 & RESERVED \\
xxxx1 & B \\
xxx10 & H \\
xx100 & S \\
x1000 & D \\
\hline
\end{tabular}
\end{verbatim}
For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Ts> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<V> Is the destination width specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(idxsizes) operand = V[n];
bits(datasizes) result;
bits(esizes) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
// imm5<4:size+1> is IGNORED
if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “imm5:Q”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

Unspecified bits in “imm5” are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**EOR (vector)**

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 1 0
```

EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];
operand1 = V[m];
operand2 = Zeros();
operand3 = Ones();
V[d] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when \textit{FEAT\_SHA3} is implemented.

### Advanced SIMD
\textbf{(FEAT\_SHA3)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | Rm | Ra | Rn | Rd |

\textbf{EOR3} \(<V_d>.16B, <V_n>.16B, <V_m>.16B, <V_a>.16B>

if \(!\text{HaveSHA3Ext}()\) then UNDEFINED;
integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
integer a = \text{UInt}(Ra);

### Assembler Symbols

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\(<V_a>\) Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

### Operation

\texttt{AArch64.CheckFPAvSIMDEnabled();}

\texttt{bits(128) \ V_m = V[m];}
\texttt{bits(128) \ V_n = V[n];}
\texttt{bits(128) \ V_a = V[a];}
\texttt{V[d] = V_n \ EOR \ V_m \ EOR \ V_a;}

### Operational information

If \texttt{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

<table>
<thead>
<tr>
<th>Q</th>
<th>imm4&lt;3&gt;</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>imm4&lt;2:0&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>imm4</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi:lo;
V[d] = concat<position+datasize-1:position>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

### Scalar half precision

((FEAT_FP16)

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FABD** <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;

### Scalar single-precision and double-precision

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sz</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FABD** <V>d>, <V>n>, <V>m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;

### Vector half precision

((FEAT_FP16)

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FABD**
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

### Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| U  |

FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

### Assembler Symbols

-Hd- Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
-Hn- Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
-Hm- Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
-V- Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

-d- Is the number of the SIMD&FP destination register, in the "Rd" field.

-n- Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

-m- Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

-Vd- Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

-T- For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

-Vn- Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRTypen fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>opc</th>
<th>ftype</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001101</td>
<td>00000010</td>
<td>10000000</td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision (ftype == 11)**
(FEAT_FP16)

```
FABS <Hd>, <Hn>
```

**Single-precision (ftype == 00)**

```
FABS <Sd>, <Sn>
```

**Double-precision (ftype == 01)**

```
FABS <Dd>, <Dn>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'  
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];

Elem[result, 0, esize] = FPAbs(operand);
V[d] = result;
```
FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision (FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------|-------|----------------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FABS <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------|-------|----------------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FABS <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\[ \text{<Vn>} \] Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision, and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Scalar single-precision and double-precision

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```
integer \( d = \text{UInt}(Rd); \)
integer \( n = \text{UInt}(Rn); \)
integer \( m = \text{UInt}(Rm); \)
integer \( \text{esize} = 32 \ll \text{UInt}(sz); \)
integer \( \text{datasize} = \text{esize}; \)
integer \( \text{elements} = 1; \)
\texttt{CompareOp} \( \text{cmp}; \)
boolean \( \text{abs}; \)

\texttt{case \text{E:U:ac of}
when '000' \( \text{cmp} = \text{CompareOp\_EQ}; \) \text{abs} = FALSE;
when '010' \( \text{cmp} = \text{CompareOp\_GE}; \) \text{abs} = FALSE;
when '011' \( \text{cmp} = \text{CompareOp\_GE}; \) \text{abs} = TRUE;
when '110' \( \text{cmp} = \text{CompareOp\_GT}; \) \text{abs} = FALSE;
when '111' \( \text{cmp} = \text{CompareOp\_GT}; \) \text{abs} = TRUE;
otherwise UNDEFINED;
\texttt{case \text{E:U:ac of}
when '010' \( \text{cmp} = \text{CompareOp\_GE}; \) \text{abs} = FALSE;
when '011' \( \text{cmp} = \text{CompareOp\_GE}; \) \text{abs} = TRUE;
when '110' \( \text{cmp} = \text{CompareOp\_GT}; \) \text{abs} = FALSE;
when '111' \( \text{cmp} = \text{CompareOp\_GT}; \) \text{abs} = TRUE;
otherwise UNDEFINED;
\texttt{case \text{E:U:ac of}
when '000' \( \text{cmp} = \text{CompareOp\_EQ}; \) \text{abs} = FALSE;
when '010' \( \text{cmp} = \text{CompareOp\_GE}; \) \text{abs} = FALSE;
when '011' \( \text{cmp} = \text{CompareOp\_GE}; \) \text{abs} = TRUE;
when '110' \( \text{cmp} = \text{CompareOp\_GT}; \) \text{abs} = FALSE;
when '111' \( \text{cmp} = \text{CompareOp\_GT}; \) \text{abs} = TRUE;
otherwise UNDEFINED;
\text{Vector half precision}
(\text{FEAT\_FP16})

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\text{Vector single-precision and double-precision}

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(eseize) element1;
bits(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in \( \text{FPCR} \), the exception results in either a flag being set in \( \text{FPSR} \), or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the \( \text{CPACR_EL1, CPTR_EL2, and CPTR_EL3} \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 1 0 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

FACGT \(<Hd>, <Hn>, <Hm>\)

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
integer \(\text{esize} = 16\);
integer \(\text{datasize} = \text{esize}\);
integer \(\text{elements} = 1\);
\(\text{CompareOp} \text{ cmp};\)

boolean \(\text{abs}\);

case E:U:ac of
  when '000' \(\text{cmp} = \text{CompareOp_EQ}; \text{abs} = \text{FALSE};\)
  when '010' \(\text{cmp} = \text{CompareOp_GE}; \text{abs} = \text{FALSE};\)
  when '011' \(\text{cmp} = \text{CompareOp_GE}; \text{abs} = \text{TRUE};\)
  when '110' \(\text{cmp} = \text{CompareOp_GT}; \text{abs} = \text{FALSE};\)
  when '111' \(\text{cmp} = \text{CompareOp_GT}; \text{abs} = \text{TRUE};\)
  otherwise UNDEFINED;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1</td>
</tr>
</tbody>
</table>

\(\text{FACGT}\)
FACGT <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Vector half precision
(FEAT_FP16)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | 1 | Rn | Rd
| U | E | ac |

FACGT <Vd>.<T>, <Vm>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Vector single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd
| U | E | ac |
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(eseize) element1;
bits(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 &\& IsMerging(fpcr);
bites(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | ftype | 1 | Rm | 0 | 0 | 1 | 0 | 1 | 0 | Rn | Rd | op |

Half-precision (ftype == 11)
(FEAT_FP16)

FADD <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FADD <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FADD <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypedef pc = FPCR[];
boolean merge = IsMerging(pc);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPAdd(operand1, operand2, pc);
V[d] = result;
```
**FADD (vector)**

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

### Half-precision

(***FEAT_FP16***)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FADD** `<Vd>..<T>, <Vn>..<T>, <Vm>..<T>`

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FADD** `<Vd>..<T>, <Vn>..<T>, <Vm>..<T>`

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer m = **UInt**(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << **UInt**(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 1 0 1 1 1 1 1 0 0 0 | sz 1 1 0 0 0 | 0 1 1 0 1 1 0 |
| Rn                | Rd              |

FADDP <V><d>, <Vn><T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;

Single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0 0</td>
<td>sz 1 1 0 0 0</td>
<td>0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>Rn</td>
<td>Rd</td>
<td></td>
</tr>
</tbody>
</table>

FADDP <V><d>, <Vn><T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize * 2;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FADD, operand, esize);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(_FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | Rm | 0 | 0 | 0 | 1 | 0 | 1 | Rn | Rd |
| U                           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 1 | 0 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
| U                           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

### Assembler Symbols

- **<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCADD

Floating-point Complex Add.
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with
the more significant element holding the imaginary part of the number and the less significant element holding the
real part of the number. Each element holds a floating-point value. It performs the following computation on the
corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is
  rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in
either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point
exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_FCMA)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | size:Q | 0  | 1  | 1  | 1  | rot:rotate | 0  | 1  | Rd  |

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>1</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = \( V[n] \);
bits(datasize) operand2 = \( V[m] \);
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
  case rot of
    when '0'
      element1 = \( \text{FPNeg}(\text{Elem}(\text{operand2}, e*2+1, esize)) \);
      element3 = \( \text{Elem}(\text{operand2}, e*2, esize) \);
    when '1'
      element1 = \( \text{Elem}(\text{operand2}, e*2+1, esize) \);
      element3 = \( \text{FPNeg}(\text{Elem}(\text{operand2}, e*2, esize)) \);
  \( \text{Elem}(\text{result}, e*2, esize) = \text{FPAdd}(\text{Elem}(\text{operand1}, e*2, esize), element1, \text{FPCR[]}) \);
  \( \text{Elem}(\text{result}, e*2+1, esize) = \text{FPAdd}(\text{Elem}(\text{operand1}, e*2+1, esize), element3, \text{FPCR[]}) \);

\( V[d] = \text{result} \);
FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the \textit{PSTATE}.\{N, Z, C, V\} flags. If the condition does not pass then the \textit{PSTATE}.\{N, Z, C, V\} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 0 1 1 1 1 0 ftype 1 | Rn  | cond  | 0 1 | Rm  | 0 | nzcv
\end{verbatim}

### Half-precision (ftype == 11) (FEAT\_FP16)

\begin{verbatim}
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
\end{verbatim}

### Single-precision (ftype == 00)

\begin{verbatim}
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
\end{verbatim}

### Double-precision (ftype == 01)

\begin{verbatim}
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
\end{verbatim}

integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;
  else
    bits(4) flags = nzcv;

Assembler Symbols

\begin{verbatim}
<Dn>  Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>  Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>  Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>  Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>  Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>  Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.
\end{verbatim}
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];

if ConditionHolds(cond) then
    flags = FPCmpare(operand1, operand2, FALSE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the `PSTATE` condition flags to N=0, Z=0, C=1, and V=1.
FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.\{N, Z, C, V\} flags. If the condition does not pass then the PSTATE.\{N, Z, C, V\} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & ftype & 1 & Rm & cond & 0 & 1 & Rn & 1 & nzcv \\
\end{array}
\]

**op**

**Half-precision (ftype == 11)**

(FEAT_FP16)

FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>

**Single-precision (ftype == 00)**

FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>

**Double-precision (ftype == 01)**

FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datasize = 16;
    else
UNDEFINED;

bits(4) flags = nzcv;

Assembler Symbols

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.
**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];

if ConditionHolds(cond) then
  flags = FPCompare(operand1, operand2, TRUE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

**Operational information**

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the `PSTATE` condition flags to N=0, Z=0, C=1, and V=1.
FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(FEAT_FP16)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm |     | 0  | 0  | 1  | 0  | 0  | 1  |     | Rd |     |

U: E: ac

FCMEQ <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = Compareop_GT; abs = TRUE;
  otherwise UNDEFINED;

Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm |     | 1  | 1  | 1  | 0  | 0  | 1  |     | Rd |     |

U: E: ac
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision
(\texttt{FEAT\_FP16})

\begin{tabular}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & \hline
Q & E & Rm & 0 & 0 & 1 & 0 & 0 & 1 & Rn & Rd
\end{tabular}

FCMEQ \texttt{(<Vd>.<T>, <Vn>.<T>, <Vm>.<T>)}

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision

\begin{tabular}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & \hline
Q & E & Rm & sz & 1 & 1 & 1 & 0 & 0 & 1 & Rn & Rd
\end{tabular}
FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(eseize) element1;
bits(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if abs then
    element1 = FPAbs(element1);
    element2 = FPAbs(element2);
  case cmp of
    when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
    when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
    when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

\[ U \]

FCMEQ <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

\[ U \]

FCMEQ <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
Vector half precision
( FEAT_FP16 )

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|--|--|--|--|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn |   |   |   |   |   |   |   |   |
| U  | op|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|--|--|--|--|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn |   |   |   |   |   |   |   |   |
| U  | op|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean testPassed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT testPassed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE testPassed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ testPassed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE testPassed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT testPassed = FPCompareGT(zero, element, FPCR[]);
        Elem[result, e, esize] = if testPassed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 0 0 1 0 | 0 0 1 0 0 1 | Rm | Rd |
| U | E | ac |

FCMGE <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 0 0 | 0 sz 1 | Rm | Rd |
| U | E | ac |
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;

Vector half precision
(_FEAT_FP16)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;

Vector single-precision and double-precision
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\texttt{CheckFPAdvSIMEnabled64();}
\texttt{bits(datasize) operand1 = V[n];}
\texttt{bits(datasize) operand2 = V[m];}

\texttt{bits(eseize) element1;}
\texttt{bits(eseize) element2;}
\texttt{boolean test\_passed;}
\texttt{FPCRType fpcr = FPCR[];}
\texttt{boolean merge = elements == 1 && IsMerging(fpcr);}
\texttt{bits(128) result = if merge then V[m] else Zeros();}

\texttt{for e = 0 to elements-1}
\texttt{element1 = Elem[operand1, e, esize];}
\texttt{element2 = Elem[operand2, e, esize];}
\texttt{if abs then}
\texttt{element1 = FPAbs(element1);}
\texttt{element2 = FPAbs(element2);}
\texttt{case cmp of}
\texttt{when CompareOp\_EQ test\_passed = FPCompareEQ(element1, element2, fpcr);}
\texttt{when CompareOp\_GE test\_passed = FPCompareGE(element1, element2, fpcr);}
\texttt{when CompareOp\_GT test\_passed = FPCompareGT(element1, element2, fpcr);}
\texttt{Elem[result, e, esize] = if test\_passed then Ones() else Zeros();}
\texttt{V[d] = result;
FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|
| 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 1 0 | Rn | Rd |
```

U   op

```
FCMGE <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|
| 0 1 1 1 1 1 1 1 1 1 0 1 | sz | 1 0 0 0 0 0 1 1 0 0 1 0 |
```

U   op

```
FCMGE <V><d>, <V><n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```
Vector half precision
( FEAT_FP16 )

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd
U | op

FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd
U | op

FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td></td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 1 0 Rm 0 0 1 0 0 1 Rn Rd</td>
</tr>
</tbody>
</table>

U E ac

FCMGT <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 sz 1 Rm 1 1 1 0 0 1 Rn Rd</td>
</tr>
</tbody>
</table>

U E ac
integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
integer esize = 32 $\ll$ $\text{UInt}(sz)$;
integer datasize = esize;
integer elements = 1;
\textit{CompareOp} cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = \textit{CompareOp\_EQ};  abs = FALSE;
  when '010' cmp = \textit{CompareOp\_GE};  abs = FALSE;
  when '011' cmp = \textit{CompareOp\_GE};  abs = TRUE;
  when '110' cmp = \textit{CompareOp\_GT};  abs = FALSE;
  when '111' cmp = \textit{CompareOp\_GT};  abs = TRUE;
  otherwise UNDEFINED;

\textbf{Vector half precision}
(\textit{FEAT\_FP16})

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rn | Rd |
| E | U  |   |   |   |   |   |   |   |   | ac |

\textbf{FCMGT} <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if \! \textit{HaveFP16Ext}() then UNDEFINED;

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
\textit{CompareOp} cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = \textit{CompareOp\_EQ};  abs = FALSE;
  when '010' cmp = \textit{CompareOp\_GE};  abs = FALSE;
  when '011' cmp = \textit{CompareOp\_GE};  abs = TRUE;
  when '110' cmp = \textit{CompareOp\_GT};  abs = FALSE;
  when '111' cmp = \textit{CompareOp\_GT};  abs = TRUE;
  otherwise UNDEFINED;

\textbf{Vector single-precision and double-precision}

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn | Rd |
| E | U  |   |   |   |   |   |   |   |   | ac |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(thesize) element1;
bits(thesize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 &amp; IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
**FCMGT (zero)**

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

### Scalar half precision

*(FEAT_FP16)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  |

**U** op

**Rn** | **Rd**

**FCMGT** $<$Hd$>$, $<$Hn$>$, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;

case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

### Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |

**U** op

**Rn** | **Rd**

**FCMGT** $<$V$>$<d$>$, $<$V$>$<n$>$, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;

case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
Vector half precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-------------------------|
| \(0|Q|0|0|1|1|1|0|1|1|1|1|0|0|0|1|1|0|0|1|0|\) | \(Rn\) | \(Rd\) |
| \(U\) | \(op\) |

FCMGT \(<Vd>.<T>, <Vn>.<T>\), \#0.0

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

integer esize = 16;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

\(\text{CompareOp}\) \(\text{comparison}\);
\text{case} \(\text{op}:U\) \text{of}
\(\text{when} \ '00' \ \text{comparison} = \text{CompareOp\_GT};\)
\(\text{when} \ '01' \ \text{comparison} = \text{CompareOp\_GE};\)
\(\text{when} \ '10' \ \text{comparison} = \text{CompareOp\_EQ};\)
\(\text{when} \ '11' \ \text{comparison} = \text{CompareOp\_LE};\)

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-------------------------|
| \(0|Q|0|0|1|1|1|1|0|1|sz|1|0|0|0|0|0|1|1|0|0|1|0|\) | \(Rn\) | \(Rd\) |
| \(U\) | \(op\) |

FCMGT \(<Vd>.<T>, <Vn>.<T>\), \#0.0

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer esize = 32 \(<\text{UInt}(sz)\);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

\(\text{CompareOp}\) \(\text{comparison}\);
\text{case} \(\text{op}:U\) \text{of}
\(\text{when} \ '00' \ \text{comparison} = \text{CompareOp\_GT};\)
\(\text{when} \ '01' \ \text{comparison} = \text{CompareOp\_GE};\)
\(\text{when} \ '10' \ \text{comparison} = \text{CompareOp\_EQ};\)
\(\text{when} \ '11' \ \text{comparison} = \text{CompareOp\_LE};\)

Assembler Symbols

\(<Hd>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<Hn>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\(<V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Complex Multiply Accumulate.
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<rotate>` Is the rotation, encoded in "rot":

```plaintext
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;
FPCRType fpcr = FPCR[];
for e = 0 to (elements DIV 2)-1
  case rot of
    when '00'
      element1 = Elem[operand2, e*2, esize];
      element2 = Elem[operand1, e*2, esize];
      element3 = Elem[operand2, e*2+1, esize];
      element4 = Elem[operand1, e*2, esize];
    when '01'
      element1 = FPNeg(Elem[operand2, e*2+1, esize]);
      element2 = Elem[operand1, e*2+1, esize];
      element3 = Elem[operand2, e*2, esize];
      element4 = Elem[operand1, e*2+1, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, e*2, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, e*2+1, esize]);
      element4 = Elem[operand1, e*2, esize];
    when '11'
      element1 = Elem[operand2, e*2+1, esize];
      element2 = Elem[operand1, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, e*2, esize]);
      element4 = Elem[operand1, e*2+1, esize];
  
  Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
  Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);
  V[d] = result;
```
**FCMLA (by element)**

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Vector

**(FEAT_FCMA)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | size | L  | M  | Rm | 0  | rot | 1  | H  | 0  | Rn | Rd |

**(size == 01)**

FCMLA `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<Ts>[<index>], #<rotate>`

**(size == 10)**

FCMLA `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<Ts>[<index>], #<rotate>`

```assembly
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index;
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;
```

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

**FCMLA (by element)**

Page 944
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in "size:H:L":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<rotate> Is the rotation, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    case rot of
        when '00'
            element1 = Elem[operand2, index*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, index*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, index*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, index*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];
    end when;
    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in \texttt{FPCR}, the exception results in either a flag being set in \texttt{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: \texttt{Scalar half precision}, \texttt{Scalar single-precision and double-precision}, \texttt{Vector half precision} and \texttt{Vector single-precision and double-precision}

**Scalar half precision**

\texttt{(FEAT\_FP16)}

\[
\begin{array}{ccccccccccccccccccc}
\hline
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & \hline
\end{array}
\]

\textbf{U} \quad \textbf{Rn} \quad \textbf{Rd}

\textbf{FCMLE <Hd>, <Hn>, #0.0}

```c
if !\texttt{HaveFP16Ext()} then UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

\texttt{CompareOp} comparison;
case op:U of
  when '00' comparison = \texttt{CompareOp\_GT};
  when '01' comparison = \texttt{CompareOp\_GE};
  when '10' comparison = \texttt{CompareOp\_EQ};
  when '11' comparison = \texttt{CompareOp\_LE};
```

**Scalar single-precision and double-precision**

\[
\begin{array}{ccccccccccccccccccc}
\hline
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & \hline
\end{array}
\]

\textbf{U} \quad \textbf{op} \quad \textbf{Rn} \quad \textbf{Rd}

\textbf{FCMLE <V<d>, <V<n>, #0.0}

```c
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 32 \ll \texttt{UInt}(sz);
integer datasize = esize;
integer elements = 1;

\texttt{CompareOp} comparison;
case op:U of
  when '00' comparison = \texttt{CompareOp\_GT};
  when '01' comparison = \texttt{CompareOp\_GE};
  when '10' comparison = \texttt{CompareOp\_EQ};
  when '11' comparison = \texttt{CompareOp\_LE};
```
Vector half precision
(FEAT_FP16)

FCMLE \<Vd>.<\text{T}>., \<Vn>.<\text{T}>., \#0.0

\begin{verbatim}
if !HaveFP16Ext() then UNDEFINED;

integer d = UNICODE(Rd);
integer n = UNICODE(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
\end{verbatim}

Vector single-precision and double-precision

\begin{verbatim}
FCMLE \<Vd>.<\text{T}>., \<Vn>.<\text{T}>., \#0.0

integer d = UNICODE(Rd);
integer n = UNICODE(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UNICODE(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
\end{verbatim}

Assembler Symbols

\begin{itemize}
    \item \texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
    \item \texttt{<Hn>} Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
    \item \texttt{<V>} Is a width specifier, encoded in "sz":
    \begin{center}
    \begin{tabular}{|c|c|}
    \hline
    sz & \texttt{<V>} \\
    \hline
    0 & S \\
    1 & D \\
    \hline
    \end{tabular}
    \end{center}
    \item \texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
    \item \texttt{<n>} Is the number of the SIMD&FP source register, encoded in the "Rn" field.
    \item \texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\end{itemize}
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
    when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
    when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
    when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
    when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
    when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMLT (zero)**

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision (FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCMLT <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

`CompareOp` comparison = `CompareOp_LT`;

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCMLT <V>d>, <V>n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

`CompareOp` comparison = `CompareOp_LT`;

### Vector half precision (FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCMLT (zero)
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[ ]);  
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[ ]);  
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[ ]);  
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[ ]);  
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[ ]);  
            Elem[result, e, esize] = if test_passed then Ones () else Zeros();

V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.\{N, Z, C, V\} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  |  | ftype | 1  | Rm | 0  | 0  | 1  | 0  | 0  |  | Rn | 0  | x  | 0  | 0  | 0  |  |  |

opc

Half-precision (ftype == 11 && opc == 00)

(FEAT_FP16)

FCMP <Hn>, <Hm>

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 01)

(FEAT_FP16)

FCMP <Hn>, #0.0

Single-precision (ftype == 00 && opc == 00)

FCMP <Sn>, <Sm>

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 01)

FCMP <Sn>, #0.0

Double-precision (ftype == 01 && opc == 00)

FCMP <Dn>, <Dm>

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 01)

FCMP <Dn>, #0.0

integer n = UInt(Rn);
integer m = UInt(Rm); // ignored when opc<0> == '1'

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datasize = 16;
    else
      UNDEFINED;
  boolean signal_all_nans = (opc<1> == '1');
  boolean cmp_with_zero = (opc<0> == '1');

**Assembler Symbols**

<\(D_n\)> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\(D_m\)> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\(H_n\)> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\(H_m\)> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<\(S_n\)> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\(S_m\)> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = \(V[n]\);
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0') else \(V[m]\);
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

**Operational information**

The IEEE 754 standard specifies that the result of a comparison is precisely one of \(<, ==, >\) or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the `PSTATE` condition flags to `N=0, Z=0, C=1, and V=1`.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE. {N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1  | Rm  | 0   | 0   | 1  | 0  | 0  | 0  | Rn  | 1  | x  | 0  | 0  | 0  | 0  | 0  | 0  | opc |
```

### Half-precision (ftype == 11 && opc == 10)
(FEAT_FP16)

```
FCMPE <Hn>, <Hm>
```

### Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 11)
(FEAT_FP16)

```
FCMPE <Hn>, #0.0
```

### Single-precision (ftype == 00 && opc == 10)

```
FCMPE <Sn>, <Sm>
```

### Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 11)

```
FCMPE <Sn>, #0.0
```

### Double-precision (ftype == 01 && opc == 10)

```
FCMPE <Dn>, <Dm>
```

### Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 11)

```
FCMPE <Dn>, #0.0
```

```plaintext
integer n = UInt(Rn);
integer m = UInt(Rm);  // ignored when opc<0> == '1'

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
data size = 16;
else
  UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```
Assembler Symbols

<Dn> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
    For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hn> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
    For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
    For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0') else V[m];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.
FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ftype</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ftype</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision (ftype == 11) (FEAT_FP16)

FCSEL <Hd>, <Hn>, <Hm>, <cond>

Single-precision (ftype == 00)

FCSEL <Sd>, <Sn>, <Sm>, <cond>

Double-precision (ftype == 01)

FCSEL <Dd>, <Dn>, <Dm>, <cond>

```plaintext
type d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype:
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;

result = if ConditionHolds(cond) then V[n] else V[m];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision to single-precision (ftype == 11 && opc == 00)

FCVT <Sd>, <Hn>

### Half-precision to double-precision (ftype == 11 && opc == 01)

FCVT <Dd>, <Hn>

### Single-precision to half-precision (ftype == 00 && opc == 11)

FCVT <Hd>, <Sn>

### Single-precision to double-precision (ftype == 00 && opc == 01)

FCVT <Dd>, <Sn>

### Double-precision to half-precision (ftype == 01 && opc == 11)

FCVT <Hd>, <Dn>

### Double-precision to single-precision (ftype == 01 && opc == 00)

FCVT <Sd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer srccsize;
integer dstsize;

if ftype == opc then UNDEFINED;

case ftype of
    when '00' srccsize = 32;
    when '01' srccsize = 64;
    when '10' UNDEFINED;
    when '11' srccsize = 16;

case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UNDEFINED;
    when '11' dstsize = 16;
```

### Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(srcsize) operand = V[n];
FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

Elem[result, 0, dstsize] = FPConvert(operand, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rd</th>
</tr>
</thead>
</table>

Half-precision to 32-bit (sf == 0 & ftype == 11) (FEAT_FP16)

FCVTAS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 & ftype == 11) (FEAT_FP16)

FCVTAS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 & ftype == 00)

FCVTAS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 & ftype == 00)

FCVTAS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 & ftype == 01)

FCVTAS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 & ftype == 01)

FCVTAS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
   when '00'
      fltsize = 32;
   when '01'
      fltsize = 64;
   when '10'
      UNDEFINED;
   when '11'
      if HaveFP16Ext() then
         fltsize = 16;
      else
         UNDEFINED;
Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Sn>` is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hn>` is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Dn>` is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, FPRounding_TIEAWAY);
X[d] = intval;
```

---

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTAS (vector)**

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision

(Feat_FP16)

<table>
<thead>
<tr>
<th>0 1 0 1 1 0 0 0 1 1 0 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
FCVTAS <Hd>, <Hn>
```

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>0 1 0 1 1 0 0 0 1 1 0 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
FCVTAS <V>d>, <V>n>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

### Vector half precision

(Feat_FP16)

<table>
<thead>
<tr>
<th>0 1 0 1 1 0 0 0 1 1 0 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
FCVTAS (vector)
```
FCVTAS $<Vd>..<T>, <Vn>..<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Assembler Symbols

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Hn>$ Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$<n>$ Is the number of the SIMD&FP source register, encoded in the "Rn" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bites(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTAU (scalar)**

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR`, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>rmode</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT_FP16)**

FCVTAU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT_FP16)**

FCVTAU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTAU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTAU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTAU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTAU <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11' UNDEFINED;
    if HaveFP16Ext() then fltsize = 16;
    else UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRTypefpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, FPRounding_TIEAWAY);
X[d] = intval;
FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(_FEAT_FP16)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 1 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 | Rn | Rd |

FCVTAU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 1 1 1 1 0 0 | sz | 1 0 0 0 0 1 1 1 0 0 1 0 | Rn | Rd |

FCVTAU <V<d>, <V<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector half precision
(_FEAT_FP16)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 Q 1 0 1 1 1 0 | 0 | 1 1 1 1 0 0 1 1 0 0 1 0 | Rn | Rd |

FCVTAU (vector)
FCVTAU $<V_d>..<T>$, $<V_n>..<T>$

if !HaveFP16Ext() then UNDEFINED;

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

integer esize = 16;
integer datasure = if $Q == '1'$ then 128 else 64;
integer elements = datasure DIV esize;

FPRounding $\text{rounding} = \text{FPRounding\_TIEAWAY}$;
boolean $\text{unsigned} = (U == '1');$

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  |

$<V_d>$, $<V_n>$

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << $\text{UInt}(sz)$;
integer datasure = if $Q == '1'$ then 128 else 64;
integer elements = datasure DIV esize;

FPRounding $\text{rounding} = \text{FPRounding\_TIEAWAY}$;
boolean $\text{unsigned} = (U == '1');$

Assembler Symbols

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Hn>$ Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>$sz$</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$<n>$ Is the number of the SIMD&FP source register, encoded in the "Rn" field.

$<V_d>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>$Q$</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>$sz$</th>
<th>$Q$</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<V_n>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
**FCVTL, FCVTL2**

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 Q 0 0 1 1 1 0 0 sz 1 0 0 0 0 1 0 1 1 1 1 0 Rn Rd
```

**FCVTL{2} <Vd>, <Ta>, <Vn>.<Tb>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
|----|---|---|---|---|---|---|---|-------|---|---|---|---|---|---|---|---|---|---|---|
| rmode | opcode |

Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT_FP16)

FCVTMS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT_FP16)

FCVTMS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTMS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTMS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTMS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTMS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDetectRounding(rmode);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;

Internal version only: isa v33.16decrl, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTMS (vector)**

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped. It has encodings from 4 classes: \textit{Scalar half precision}, \textit{Scalar single-precision and double-precision}, \textit{Vector half precision} and \textit{Vector single-precision and double-precision}.

### Scalar half precision

\textit{(FEAT\_FP16)}

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0</td>
<td>o2</td>
<td>o1</td>
</tr>
</tbody>
</table>

\textbf{FCVTMS} <Hd>, <Hn>

\[\text{if } \! \text{HaveFP16Ext}() \text{ then UNDEFINED;}\]

integer \(d = \text{UInt}(Rd);\)
integer \(n = \text{UInt}(Rn);\)

integer esize = 16;
integer datasize = esize;
integer elements = 1;

\[\text{FPRounding } \text{rounding} = \text{FPDecodeRounding}(o1:o2);\]
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

\textit{(FEAT\_FP16)}

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0</td>
<td>sz</td>
<td>o1</td>
</tr>
</tbody>
</table>

\textbf{FCVTMS} <V>d>, <V>n>

integer \(d = \text{UInt}(Rd);\)
integer \(n = \text{UInt}(Rn);\)

integer esize = 32 \(<\text{UInt}(sz);\)
integer datasize = esize;
integer elements = 1;

\[\text{FPRounding } \text{rounding} = \text{FPDecodeRounding}(o1:o2);\]
boolean unsigned = (U == '1');

### Vector half precision

\textit{(FEAT\_FP16)}

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 1 1 0</td>
<td>o2</td>
<td>o1</td>
</tr>
</tbody>
</table>
FCVTMS <Vd>, <T>, <Vn>, <T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-----------------|-----------------|------------------|
|                  | sz              |                  |                  |
| U                | 0               | 1               | 0                |
| 0                | 0               | 0               | 0                |
| 0                | 0               | 0               | 1                |
| 0                | 0               | 0               | 1                |
| 1                | 0               | 1               | 0                |
| 1                | 0               | 1               | 1                |
| 1                | 1               | 1               | 0                |
| 0                | 0               | 0               | 0                |
| 0                | 0               | 0               | 1                |
| 0                | 0               | 0               | 1                |
| 1                | 0               | 1               | 0                |
| 1                | 0               | 1               | 1                |
| 1                | 1               | 1               | 0                |

<Asm>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTypel fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPTofixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTMU (scalar)**

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode | opcode |

**Half-precision to 32-bit (sf == 0 & ftype == 11)**

**(FEAT_FP16)**

```
FCVTMU <Wd>, <Hn>
```

**Half-precision to 64-bit (sf == 1 & ftype == 11)**

**(FEAT_FP16)**

```
FCVTMU <Xd>, <Hn>
```

**Single-precision to 32-bit (sf == 0 & ftype == 00)**

```
FCVTMU <Wd>, <Sn>
```

**Single-precision to 64-bit (sf == 1 & ftype == 00)**

```
FCVTMU <Xd>, <Sn>
```

**Double-precision to 32-bit (sf == 0 & ftype == 01)**

```
FCVTMU <Wd>, <Dn>
```

**Double-precision to 64-bit (sf == 1 & ftype == 01)**

```
FCVTMU <Xd>, <Dn>
```

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPDencodeRounding(rmode);
```
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTMU (vector)**

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

**Scalar half precision**

**(FEAT_FP16)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |   |   |   |   |   |   |

U    o2    o1

**FCVTMU <Hd>, <Hn>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecimalRounding(o1:o2);
boolean unsigned = (U == '1');
```

**Scalar single-precision and double-precision**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |

U    o2    o1

**FCVTMU <V><d>, <V><n>**

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecimalRounding(o1:o2);
boolean unsigned = (U == '1');
```

**Vector half precision**

**(FEAT_FP16)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |

U    o2    o1

**FCVTMU (vector)**

Page 980
FCVTMU <Vd>.,<T>, <Vn>.,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FP_Rounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(dataSize) operand = V[n];

bits(eSize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
**FCVTN, FCVTN2**

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

### Assembler Symbols

- **integer d = UInt(Rd);**
- **integer n = UInt(Rn);**
- **integer esize = 16 << UInt(sz);**
- **integer datasize = 64;**
- **integer part = UInt(Q);**
- **integer elements = datasize DIV esize;**

#### 2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Tb>** Is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Ta>** Is an arrangement specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR[]);

Vpart[d, part] = result;
```

Internal version only: isa v31.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| sf | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode | opcode |
```

**Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT_FP16)**

FCVTNS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT_FP16)**

FCVTNS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTNS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTNS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTNS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTNS <Xd>, <Dn>

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);

type intsize = if sf == '1' then 64 else 32;
type fltsize;

FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPRounding(rmode);
```
**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
FPCRTYPE fpcr = FPCR[];
bis(FLTSIZE) fltval;
bis(INTSIZE) intval;

fltval = V[n];
intval = FPTofixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
```

*Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.*
FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

U | o2 | o1

FCVTNS <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDetectRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

U | o2 | o1

FCVTNS <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDetectRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

U | o2 | o1
FCVTNS <Vd>..<T>, <Vn>..<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |   |
| o1 | o2 | Rn | Rd |

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
**FCVTNU (scalar)**

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision to 32-bit (sf == 0 && ftype == 11)**

(Feat_FP16)

FCVTNU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**

(Feat_FP16)

FCVTNU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTNU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTNU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTNU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTNU <Xd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then fltsize = 16;
    else UNDEFINED;

rounding = FPDetectRounding(rmode);
```
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FFEAT_FP16)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| 0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 0 1 0 |
| Rn | Rd |
| U  | o2 | o1 |

FCVTNU <Hd>, <Hn>
```

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| 0 1 1 1 1 1 1 0 0 | sz 1 0 0 0 0 1 1 0 1 0 1 0 |
| Rn | Rd |
| U  | o2 | o1 |

FCVTNU <V>d>, <V>n>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(FFEAT_FP16)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| 0 Q 1 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 0 1 0 |
| Rn | Rd |
| U  | o2 | o1 |
```

FCVTNU (vector)
FCVTNU `<Vd>`..<`T`>, `<Vn>`..<`T`

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FP_Rounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

|       | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| `U`   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| `o2`  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| `Rd`  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| `Rn`  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FCVTNU `<Vd>`..<`T`>, `<Vn>`..<`T`

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FP_Rounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>` is a width specifier, encoded in "sz"

<table>
<thead>
<tr>
<th><code>sz</code></th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>` is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in "Q"

<table>
<thead>
<tr>
<th><code>Q</code></th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q"

<table>
<thead>
<tr>
<th><code>sz</code></th>
<th><code>Q</code></th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(datasize) operand = V[n];}

\texttt{bits(esize) element;}
\texttt{FPCRTypen fpcr = FPCR[;];}
\texttt{boolean merge = elements == 1 \&\& IsMerging(fpcr);}
\texttt{bits(128) result = if merge then V[d] else Zeros();}

\texttt{for e = 0 to elements-1}
  \texttt{element = Elem[operand, e, esize];}
  \texttt{Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);}
\texttt{V[d] = result;}

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTPS (scalar)**

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR`, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

---

### Half-precision to 32-bit (sf == 0 & ftype == 11) (FEAT_FP16)

```
FCVTPS <Wd>, <Hn>
```

### Half-precision to 64-bit (sf == 1 & ftype == 11) (FEAT_FP16)

```
FCVTPS <Xd>, <Hn>
```

### Single-precision to 32-bit (sf == 0 & ftype == 00)

```
FCVTPS <Wd>, <Sn>
```

### Single-precision to 64-bit (sf == 1 & ftype == 00)

```
FCVTPS <Xd>, <Sn>
```

### Double-precision to 32-bit (sf == 0 & ftype == 01)

```
FCVTPS <Wd>, <Dn>
```

### Double-precision to 64-bit (sf == 1 & ftype == 01)

```
FCVTPS <Xd>, <Dn>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPDetectRounding(rmode);
```
Assemble Symbol

\begin{align*}
&Wd & \text{Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
&Xd & \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.} \\
&Sn & \text{Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.} \\
&Hn & \text{Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.} \\
&Dn & \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\end{align*}

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
\end{verbatim}

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTPS (vector)**

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision (FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

### FCVTPS <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 1 sz 1 0 0 0 0 1 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

### FCVTPS <V>d>, <V<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

### Vector half precision (FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

### FCVTPS (vector)
FCVTPS <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTypex fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
     element = Elem[operand, e, esize];
     Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTPU (scalar)**

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR`, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision to 32-bit (sf == 0 && ftype == 11)**
*(FEAT_FP16)*

```
FCVTPU <Wd>, <Hn>
```

**Half-precision to 64-bit (sf == 1 && ftype == 11)**
*(FEAT_FP16)*

```
FCVTPU <Xd>, <Hn>
```

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

```
FCVTPU <Wd>, <Sn>
```

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

```
FCVTPU <Xd>, <Sn>
```

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

```
FCVTPU <Wd>, <Dn>
```

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

```
FCVTPU <Xd>, <Dn>
```

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then fltsize = 16;
    else UNDEFINED;
  rounding = FPDetectRounding(rmode);
```
Assembler Symbols

<Wd>   Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>   Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>   Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>   Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>   Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;
FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(Feat_FP16)

```
0 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| U | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Rd |
```

FCVTPU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDEncodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

```
0 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| U | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rd |
```

FCVTPU <V>d>, <V<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDEncodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision

(Feat_FP16)

```
0 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| U | 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | Rd |
```

FCVTPU (vector)
FCVTPU `<Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FP Rounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |
| O2 | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(32) element;
FPCRTYPE  fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bv128 result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| Rd | Rn |

FCVTXN <Vb><d>, <Va><n>

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == ‘0’ then UNDEFINED;
integer esize = 32;
iconteger datasize = esize;
iconteger elements = 1;
iconteger part = 0;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| Rd | Rn |

FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == ‘0’ then UNDEFINED;
iconteger esize = 32;
iconteger datasize = 64;
iconteger elements = 2;
iconteger part = UInt(Q);
Assembler Symbols

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 &\& IsMerging(fpcr);
bv128 result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], fpcr, FPRounding_ODD);

if merge then
    V[d] = result;
else
    Vpart[d, part] = Elem[result, 0, datasize];
FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT_FP16)

FCVTZS <Wd>, <Hn>, <fbits>

Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT_FP16)

FCVTZS <Xd>, <Hn>, <fbits>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTZS <Wd>, <Sn>, <fbits>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTZS <Xd>, <Sn>, <fbits>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTZS <Wd>, <Dn>, <fbits>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTZS <Xd>, <Dn>, <fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".
For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPAdvSIMDEnabled64();
FPCRTYPE fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, frachits, FALSE, fpcr, FPRounding_ZERO);
X[d] = intval;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTZS (scalar, integer)**

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rd</th>
</tr>
</thead>
</table>

**Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT_FP16)**

FCVTZ <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT_FP16)**

FCVTZ <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZ <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZ <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZ <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTZ <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPDetectRounding(rmode);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRTypefpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|-----------------|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | != 0000 | immb | 1 | 1 | 1 | 1 | 1 | Rn | Rd |

U \text{ immh}

FCVTZS <V><d>, <V><n>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 1 | != 0000 | immb | 1 | 1 | 1 | 1 | 1 | Rn | Rd |

U \text{ immh}

FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
b Yanis (128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
```

FCVTZS (vector, fixed-point)
**FCVTZS (vector, integer)**

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

**Scalar half precision**

**FEAT_FP16**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**FCVTZS <Hd>, <Hn>**

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

**Scalar single-precision and double-precision**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**FCVTZS <V>d>, <V>n>**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

**Vector half precision**

**FEAT_FP16**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**FCVTZS (vector, integer)**
FCVTZS <Vd>.<T>, <Vn>.<T>

if ![HaveFP16Ext]() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTyp fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision to 32-bit (sf == 0 && ftype == 11) (FEAT_FP16)

FCVTU <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit (sf == 1 && ftype == 11) (FEAT_FP16)

FCVTU <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTU <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTU <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTU <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTU <Xd>, <Dn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end;
if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".

For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPTofixed(fltval, frachbits, TRUE, fpcr, FPRounding_ZERO);
X[d] = intval;
**FCVTZU (scalar, integer)**

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | ft | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode | opcode |

**Half-precision to 32-bit (sf == 0 & ftype == 11)**

(FEAT_FP16)

FCVTZU <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 & ftype == 11)**

(FEAT_FP16)

FCVTZU <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 & ftype == 00)**

FCVTZU <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 & ftype == 00)**

FCVTZU <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 & ftype == 01)**

FCVTZU <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 & ftype == 01)**

FCVTZU <Xd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
rounding = FPDetectRounding(rmode);
```
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRTypet fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTZU (vector, fixed-point)**

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1, CPTR_EL2**, and **CPTR_EL3** registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>immb</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCVTZU <V><d>, <V><n>, #<fbits>**

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

### Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!=</td>
<td>0000</td>
<td>immb</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

**Assembler Symbols**

`<V>` Is a width specifier, encoded in “immh”:

---

**FCVTZU (vector, fixed-point)**
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bign(size) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bign(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTZU (vector, integer)**

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped. It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision
(Feat_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>02 o2 o1</td>
</tr>
</tbody>
</table>

**FCVTZU <Hd>, <Hn>**

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datatype = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 0 0 0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>02 o2 o1</td>
</tr>
</tbody>
</table>

**FCVTZU <V<d>, <V<n>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datatype = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

### Vector half precision
(Feat_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 0 1 1 1 1 1 0 0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>02 o2 o1</td>
</tr>
</tbody>
</table>
FCVTZU \(<Vd>.<T>, <Vn>.<T>\)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<Vd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Assembler Symbols

\(<Hd>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<Hn>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\(<V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
**FDIV (scalar)**

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPSCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 1     | 1     | 1     | 0     | ftype | 1     | Rm    | 0     | 0     | 1     | 1     | 0     | Rn    | Rd    |
```

**Half-precision (ftype == 11)**

(FEAT_FP16)

```c
FDIV <Hd>, <Hn>, <Hm>
```

**Single-precision (ftype == 00)**

```c
FDIV <Sd>, <Sn>, <Sm>
```

**Double-precision (ftype == 01)**

```c
FDIV <Dd>, <Dn>, <Dm>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

inguar esize;
(case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
)

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[ result, 0, esize] = FDIV(operand1, operand2, FPCR[]);

V[d] = result;
FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Feat_FP16)

```
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 1 | 1 | Rn | Rd
```

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Single-precision and double-precision**

```
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 1 | 1 | 1 | Rn | Rd
```

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

```
| Q | <T>
|---|---
| 0 | 4H
| 1 | 8H
```

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

FDIV (vector)
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPDiv(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo $2^{32}$, as held in a 32-bit signed integer. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit (FEAT_JSCVT)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | ftype | rmode | opcode |

FJCVTZS <Wd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveFJCVTZSExt() then UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(64) fltval;
bits(32) intval;

bit Z;
fltval = V[n];
(intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);
PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;
FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 1 ftype 0 | Rm | 0 | Ra | Rn | Rd

Half-precision (ftype == 11)
(FEAT_FP16)

FMADD <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (ftype == 00)

FMADD <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (ftype == 01)

FMADD <Dd>, <Dn>, <Dm>, <Da>

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
 case ftype of
   when '00' esize = 32;
   when '01' esize = 64;
   when '10' UNDEFINED;
   when '11'
      if HaveFP16Ext() then
         esize = 16;
      else
         UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypetype fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMAX (scalar)**

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>ftype</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Half-precision (ftype == 11)

**(FEAT_FP16)**

FMAX <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMAX <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMAX <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

---

**Assembler Symbols**

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Hm>** Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Sm>** Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\text{CheckFPAdvSIMDEnabled64}();
bits(\text{esize}) \text{operand1} = V[n];
bits(\text{esize}) \text{operand2} = V[m];

\text{FPCRTypen} fpcr = \text{FPCR}[];
\text{boolean merge} = \text{IsMerging}(fpcr);
bits(128) \text{result} = \text{if merge then } V[n] \text{ else } \text{Zeros}();

\text{Elem}[\text{result}, 0, \text{esize}] = \text{FPMax}(\text{operand1}, \text{operand2}, \text{fpcr});
V[d] = \text{result};
FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FFT_FP16)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMAX <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMAX <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | Rm | 0  | 1  | 1  | 0  | 1  | 0  | Rn  | Rd  | op |

Half-precision (ftype == 11) (FEAT_FP16)

FMAXNM <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FMAXNM <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FMAXNM <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<DN> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<DM> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypelfcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMAXNM (vector)**

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to `FMAX (scalar)`.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR`, or a synchronous exception being generated. For more information, see `Floating-point exception traps`.

Depending on the settings in the `CPACR_EL1, CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: `Half-precision` and `Single-precision and double-precision`

### Half-precision

**(FEAT_F16)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 0  | 1  | Rd |
| U  | a  |

**FMAXNM `<Vd>..<T>, `<Vn>.<T>, `<Vm>.<T>`**

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
icnter n = UInt(Rn);
icnter m = UInt(Rm);
icnter esize = 16;
icnter datasize = if Q == '1' then 128 else 64;
icnter elements = datasize DIV esize;
boolen pair = (U == '1');
icnter minimum = (a == '1');
```

### Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 0  | 0  | 1  | Rd |
| U  | o1 |

**FMAXNM `<Vd>..<T>, `<Vn>.<T>, `<Vm>.<T>`**

```plaintext
integer d = UInt(Rd);
icnter n = UInt(Rn);
icnter m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
icnter esize = 32 << UInt(sz);
icnter datasize = if Q == '1' then 128 else 64;
icnter elements = datasize DIV esize;
boolen pair = (U == '1');
icnter minimum = (o1 == '1');
```

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in "Q": 

---

FMAXNM (vector)  
Page 1038
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEATURE_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 0 1 1 1 1 0 0 | sz | 1 1 0 0 0 | 0 1 1 0 0 | 1 0 | Rn | Rd |

FMAXNMP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 1 1 1 1 1 0 0 | sz | 1 1 0 0 0 | 0 1 1 0 0 | 1 0 | Rn | Rd |

FMAXNMP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision (FEAT_FP16)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|   0 |  Q  | 1   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rm  |   0  | 0   | 0   | 0   | 1   | Rn  |   Rd |

U = a
```

**FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

### Single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|   0 |  Q  | 1   | 0   | 1   | 1   | 0   | 0   | 1   | 0   | Rm  | 1   | 1   | 0   | 0   | 0   | 1   | Rn  |   Rd |

U = o1
```

**FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the first SIMD\&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD\&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];

bits(datasize) operand2 = V[m];

bits(datasize) result;

bits(2*datasize) concat = operand2:operand1;

bits(esize) element1;

bits(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d] = result;
FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

( FEAT_FP16)

| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

**FMAXNMV** <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

**Single-precision and double-precision**

| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

**FMAXNMV** <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;

**Assembler Symbols**

< V >     For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th></th>
<th>&lt; V &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>     Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn>    Is the name of the SIMD&FP source register, encoded in the "Rn" field.
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
b_bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision (FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

FMAXP \(<V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

FMAXP \(<V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;

### Assembler Symbols

\(<V>\) For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(_FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|   0 | 0   | 1   | 1   | 1   | 0   | 0   | 1   | 0   | Rm  | 0   | 0   | 1   | 1   | 0   | 1   | Rn  | 0   | Rd  |
|  U  |  o1 |

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|   0 | 0   | 1   | 1   | 1   | 0   | 0   | sz  | 1   | Rm  | 1   | 1   | 1   | 0   | 1   | Rn  | 0   | Rd  |
|  U  |  o1 |

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

FMAXP (vector)  Page 1048
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    
    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d] = result;
```
FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision (FEAT_FP16)

![Half-precision encoding](image)

FMAXV <V><d>, <Vn>.<T>

```plaintext
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

### Single-precision and double-precision

![Single-precision encoding](image)

FMAXV <V><d>, <Vn>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q != '01' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>For the half-precision variant: is the destination width specifier, H.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:</td>
</tr>
<tr>
<td>sz</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

| <d> | Is the number of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Vn> | Is the name of the SIMD&FP source register, encoded in the "Rn" field. |
| <T> | For the half-precision variant: is an arrangement specifier, encoded in “Q”: |
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
## FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (ftype == 11)

\texttt{(FEAT\_FP16)}

```
FMIN <Hd>, <Hn>, <Hm>
```

### Single-precision (ftype == 00)

```
FMIN <Sd>, <Sn>, <Sm>
```

### Double-precision (ftype == 01)

```
FMIN <Dd>, <Dn>, <Dm>
```

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);

integer esize;
\texttt{case ftype of}
  \texttt{when '00' esize = 32;}
  \texttt{when '01' esize = 64;}
  \texttt{when '10' UNDEFINED;}
  \texttt{when '11'}
    \texttt{if HaveFP16Ext() then}
      \texttt{esize = 16;}
    \texttt{else}
      \texttt{UNDEFINED;}

### Assembler Symbols

- \texttt{<Dd>} is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Dn>} is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Dm>} is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Hd>} is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Hn>} is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Hm>} is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Sd>} is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Sn>} is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Sm>} is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypedef pcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPMin(operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMIN (vector)**

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: *Half-precision* and *Single-precision and double-precision*

### Half-precision (FEAT_FP16)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FMIN <Vd>..<T>, <Vn>..<T>, <Vm>..<T>**

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FMIN <Vd>..<T>, <Vn>..<T>, <Vm>..<T>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMINNM (scalar)**

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar). This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Hexadecimal register encoding](image)

- **Half-precision (ftype == 11)**
  (FEAT_FP16)
  ```
  FMINNM <Hd>, <Hn>, <Hm>
  ```

- **Single-precision (ftype == 00)**
  ```
  FMINNM <Sd>, <Sn>, <Sm>
  ```

- **Double-precision (ftype == 01)**
  ```
  FMINNM <Dd>, <Dn>, <Dm>
  ```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bites(esize) operand1 = V[n];
bites(esize) operand2 = V[m];

FPCRTypencr = FPCR[];
boolean merge = IsMerging(fpcr);
bites(128) result = if merge then V[n] else Zeros();

Elem(result, 0, esize) = FPMinNum(operand1, operand2, fpcr);
V[d] = result;
**FMINNM (vector)**

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision (FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | a |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | ol |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (ol == '1');

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

---

**Page 1058**
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision (FEAT_FP16)**

![Encoding](image)

FMINNMP <V><d>, <Vn>.

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

**Single-precision and double-precision**

![Encoding](image)

FMINNMP <V><d>, <Vn>.

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

**Assembler Symbols**

- `<V>`: For the half-precision variant: is the destination width specifier, encoded in “sz”:
  
<table>
<thead>
<tr>
<th><code>sz</code></th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
  
  For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th><code>sz</code></th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: Half-precision and Single-precision and double-precision.

Half-precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| U | a |

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| U | o1 |

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to `FMIN (scalar)`.

This instruction can generate a floating-point exception. Depending on the settings in `FPCR`, the exception results in either a flag being set in `FPSR` or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision.

### Half-precision
(_FEAT_FP16)

```
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0
```

#### FMINNMV `<V><d>`, `<Vn>`. `<T>`

```
if !haveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

### Single-precision and double-precision

#### FMINNMV `<V><d>`, `<Vn>`. `<T>`

```
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q != '01' then UNDEFINED; // .4S only
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

### Assembler Symbols

- `<V>` For the half-precision variant: is the destination width specifier, H.
- `<V>` For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

```
0 | 5 | <V>
```

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision.

Half-precision

(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
| 0 1 0 1 1 1 1 0 1 sz 1 1 0 0 0 0 0 1 1 1 1 1 0 |

Rn  Rd

FMINP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 1 0 1 sz 1 1 0 0 0 0 0 1 1 1 1 1 0 |

Rn  Rd

FMINP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

V[d] = Reduce(ReduceOp_FMIN, operand, esize);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(FEATURE_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |

**FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | Rn | Rd |

**FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

---

FMINP (vector)  Page 1068
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Feat_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | Rd | | Rn |

FMINV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | Rd | | Rn |

FMINV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q !='01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;

**Assembler Symbols**

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR} or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

\begin{verbatim}
Scalar, half-precision
(\textit{FEAT\_FP16})

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|-----------------|
| 0  1  0  1  1  1  1  1  0  0 | L   | M   | Rm   | 0  0  0  1 | H   | 0 | Rn | Rd |

FMLA <Hd>, <Hn>, <Vm>.H[<index>]

if \texttt{!HaveFP16Ext()} then UNDEFINED;

integer idxdsize = if \texttt{H} == '1' then 128 else 64;
integer n = \texttt{UInt(Rn)};
integer m = \texttt{UInt(Rm)};
integer d = \texttt{UInt(Rd)};
integer index = \texttt{UInt(H:L:M)};

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (\texttt{o2} == '1');

Scalar, single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|-----------------|
| 0  1  0  1  1  1  1  1  1  1 | sz | L   | M   | Rm   | 0  0  0  1 | H   | 0 | Rn | Rd |

FMLA <V<d>, <V<n>, <Vm>.<Ts>[<index>]

integer idxdsize = if \texttt{H} == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = \texttt{UInt(H:L)};
    when '10' index = \texttt{UInt(H)};
    when '11' UNDEFINED;

integer d = \texttt{UInt(Rd)};
integer n = \texttt{UInt(Rn)};
integer m = \texttt{UInt(Rmhi:Rm)};

integer esize = 32 << \texttt{UInt(sz)};
integer datasize = esize;
integer elements = 1;
boolean sub_op = (\texttt{o2} == '1');
\end{verbatim}
Vector, half-precision

(\texttt{FEAT\_FP16})

\begin{Verbatim}
0 0 0 1 1 1 0 0 L M Rm 0 0 0 1 H 0 Rn Rd o2
\end{Verbatim}

\textbf{FMLA} \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]}\textbf{FMLA} \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]}

\begin{verbatim}
if \texttt{!HaveFP16Ext()} then \texttt{UNDEFINED};
integer idxdsize = if \texttt{H == '1'} then 128 else 64;
integer n = \texttt{UInt}(\texttt{Rn});
integer m = \texttt{UInt}(\texttt{Rm});
integer d = \texttt{UInt}(\texttt{Rd});
integer index = \texttt{UInt}(\texttt{H:L:M});

integer esize = 16;
integer datasize = if \texttt{Q == '1'} then 128 else 64;
integer elements = datasize \texttt{DIV} esize;
boolean sub_op = (\texttt{o2 == '1'});
\end{verbatim}

Vector, single-precision and double-precision

\begin{Verbatim}
0 0 0 1 1 1 1 1 sz L M Rm 0 0 0 1 H 0 Rn Rd o2
\end{Verbatim}

\textbf{FMLA} \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]}\textbf{FMLA} \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]}

\begin{verbatim}
integer idxdsize = if \texttt{H == '1'} then 128 else 64;
integer index;
bit Rmhi = M;
case \texttt{sz:L} of
  when '0x' index = \texttt{UInt}(\texttt{H:L});
  when '10' index = \texttt{UInt}(\texttt{H});
  when '11' \texttt{UNDEFINED};

integer d = \texttt{UInt}(\texttt{Rd});
integer n = \texttt{UInt}(\texttt{Rn});
integer m = \texttt{UInt}(\texttt{Rmhi:Rm});

if \texttt{sz:Q == '10'} then \texttt{UNDEFINED};
integer esize = 32 \texttt{<<} \texttt{UInt}(\texttt{sz});
integer datasize = if \texttt{Q == '1'} then 128 else 64;
integer elements = datasize \texttt{DIV} esize;
boolean sub_op = (\texttt{o2 == '1'});
\end{verbatim}

\textbf{Assembler Symbols}

\begin{verbatim}
\texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Hn>} Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
\texttt{<V>} Is a width specifier, encoded in "sz":
\begin{verbatim}
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
\end{verbatim}
\texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<n>} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

FMLA (by element)
For the half-precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{c|c}
Q & <T> \\
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

\[
\begin{array}{c|c|c}
Q & sz & <T> \\
0 & 0 & 2S \\
0 & 1 & RESERVED \\
1 & 0 & 4S \\
1 & 1 & 2D \\
\end{array}
\]

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

\[
\begin{array}{c|c}
sz & <Ts> \\
0 & 5 \\
1 & D \\
\end{array}
\]

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

\[
\begin{array}{c|c|c}
sz & L & <index> \\
0 & x & H:L \\
1 & 0 & H \\
1 & 1 & RESERVED \\
\end{array}
\]

Operation

```
CheckFPAdvSIMDEnabled64();
b bits(datasize) operand1 = V[n];
b bits(idxdsize) operand2 = V[m];
b bits(datasize) operand3 = V[d];
b bits(esize) element1;
b bits(esize) element2 = Elem[operand2, index, esize];
FPCRTypeth fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
b bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);
V[d] = result;
```
FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(FEAT_FP16)

```
0  Q  0  0  1  1  0  0  0  1  0  Rm  0  0  0  1  1  Rn  Rd
a
```

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');

**Single-precision and double-precision**

```
0  Q  0  0  1  1  0  0  sz  1  Rm  1  1  0  0  1  1  Rn  Rd
op
```

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

**Note**

`ID_AA64ISAR0_EL1.FHM` indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

**FMLAL**

(Feats_FHM)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0               | 0     | 0     | 1     | 1     | 1     | 1     | 1     | 0     | L     | M     | Rm    | 0     | 0     | 0     | 0     | H     | 0     | Rn    | Rd    |       |       |       |       |       |       |       |
```

**FMLAL** `<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[index]`

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

**FMLAL2**

(Feats_FHM)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0               | 0     | 1     | 0     | 1     | 1     | 1     | 1     | 1     | 0     | L     | M     | Rm    | 1     | 0     | 0     | 0     | H     | 0     | Rn    | Rd    |       |       |       |       |       |       |       |
```

FMLAL, FMLAL2 (by element)
FMLAL2 <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.H<index>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);  // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd>    Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta>    Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn>    Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb>    Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

<Vm>    Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<index> Is the element index, encoded in the "H:L:M" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[ ]);
V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(FEAT_FHM)

```
0 0 0 0 0 1 1 0 0 0 1 | Rm | 1 1 1 0 | 1 1 | Rn | Rd
S  sz
```

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFp32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

FMLAL2

(FEAT_FHM)

```
0 0 0 1 0 1 1 0 0 0 1 | Rm | 1 1 0 0 1 1 | Rn | Rd
S  sz
```

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFp32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
**Assembler Symbols**

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Ta>**
  - Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Tb>**
  - Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

- **<Vm>**
  - Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision, and Vector, single-precision and double-precision

Scalar, half-precision
(_FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 0 1 1 1 1 1 0 0 | L | M | Rm | 0 1 0 1 | H | 0 | Rn | Rd | o2 |

FMLS <Hd>, <Hn>, <Vm>.H[index]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Scalar, single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 0 1 1 1 1 1 1 | sz | L | M | Rm | 0 1 0 1 | H | 0 | Rn | Rd | o2 |

FMLS <V<d>, <V<n>, <Vm>.<Ts>[index]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
Vector, half-precision
(FEAT_FP16)

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Vector, single-precision and double-precision

FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<V>  Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n>  Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

\[
\begin{array}{c|c}
Q & \text{T} \\
\hline
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

\[
\begin{array}{c|c}
Q & \text{sz} \\
\hline
0 & 0 \\
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{array}
\]

<\text{n}>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}>
For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<\text{s}>
Is an element size specifier, encoded in “sz”:

\[
\begin{array}{c|c}
sz & \text{s} \\
\hline
0 & 5 \\
1 & 0 \\
\end{array}
\]

<\text{i}>
For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

\[
\begin{array}{c|c|c}
sz & L & \text{i} \\
\hline
0 & x & \text{H:L} \\
1 & 0 & \text{H} \\
1 & 1 & \text{RESERVED} \\
\end{array}
\]

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMLS (vector)**

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

### Half-precision (FEAT_FP16)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0 Q 0 0 1 1 1 0 | 1 1 1 | 1 0 | Rm | 0 0 0 0 1 1 | Rn | Rd |
```

**FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

```plaintext
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (a == '1');
```

### Single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0 Q 0 0 1 1 1 0 | 1 1 0 | 0 1 1 | sz | 1 1 0 0 1 1 | Rn | Rd |
```

**FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (op == '1');
```

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(FEAT_FHM)

```
0 | Q | O | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 0 | 1 | 0 | 0 | H | 0 | Rn | Rd
```

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[index]

```java
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint('0':Rm); // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = Uint(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2
(FEAT_FHM)

```
0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 1 | 1 | 0 | 0 | H | 0 | Rn | Rd
```

FMLSL, FMLSL2 (by element)
FMLSL2 <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.H[index]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);     // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<index> Is the element index, encoded in the "H:L:M" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize DIV 2];
  if sub_op then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the 
vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates 
the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round 
the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception 
results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see 
Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and 
Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to 
support it.

Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL

( FEAT_FHM )

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd |
| S | sz |

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

FMLSL2

( FEAT_FHM )

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | Rm | 1 | 1 | 0 | 0 | 1 | 1 | Rn | Rd |
| S | sz |

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
sf 0 0 1 1 1 0 ftype 1 0 x 1 1 x 0 0 0 0 0 0 | Rn | Rd
  rmode  opcode
```
Half-precision to 32-bit (sf == 0 && ftype == 11 && rmode == 00 && opcode == 110) (FEAT_FP16)

FMOV <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11 && rmode == 00 && opcode == 110) (FEAT_FP16)

FMOV <Xd>, <Hn>

32-bit to half-precision (sf == 0 && ftype == 11 && rmode == 00 && opcode == 111) (FEAT_FP16)

FMOV <Hd>, <Wn>

32-bit to single-precision (sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)

FMOV <Sd>, <Wn>

Single-precision to 32-bit (sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)

FMOV <Wd>, <Sn>

64-bit to half-precision (sf == 1 && ftype == 11 && rmode == 00 && opcode == 111) (FEAT_FP16)

FMOV <Hd>, <Xn>

64-bit to double-precision (sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)

FMOV <Dd>, <Xn>

64-bit to top half of 128-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)

FMOV <Vd>.D[1], <Xn>

Double-precision to 64-bit (sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)

FMOV <Xd>, <Dn>

Top half of 128-bit to 64-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 110)

FMOV <Xd>, <Vn>.D[1]
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPFPRounding rounding;
boolean unsigned;
integer part;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    if opcode<2:1>:rmode != '1 01' then UNDEFINED;
    fltsize = 128;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
    case opcode<2:1>:rmode of
      when '00 xx'    // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
      when '01 00'    // [US]CVTF
        rounding = FPRoundingMode(FPCR[]);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
      when '10 00'    // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
      when '11 00'    // FMOV
        if fltsize != 16 && fltsize != intsize then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
      when '11 01'    // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp MOV_ItoF else FPConvOp MOV_FtoI;
        part = 1;
        fltsize = 64;    // size of D[1] is 64
      when '11 11'    // FJCVTZS
        if !HaveFJCVTZSExt() then UNDEFINED;
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
      otherwise
        UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```java
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
case op of
  when FPConvOp_CVT_FtoI
    fltval = V[n];
    intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding);
    X[d] = intval;
  when FPConvOp_CVT_ItoF
    fltval = if merge then V[d] else Zeros();
    Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, unsigned, fpcr, rounding);
    V[d] = fltval;
  when FPConvOp_MOV_FtoI
    fltval = Vpart[n, part];
    intval = ZeroExtend(fltval, intsize);
    X[d] = intval;
  when FPConvOp_MOV_ItoF
    fltval = X[n];
    intval = intval<&size-1:0>;
    Vpart[d, part] = fltval;
  when FPConvOp_CVT_FtoI_JS
    bit Z;
    fltval = V[n];
    (intval, Z) = FPToFixedJS(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 | Rd
| opc
```

Half-precision (ftype == 11) (FEAT_FP16)

FMOV <Hd>, <Hn>

Single-precision (ftype == 00)

FMOV <Sd>, <Sn>

Double-precision (ftype == 01)

FMOV <Dd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[n];

Elem(Zeros(), 0, esize) = operand;
V[d] = Zeros();
```
FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (ftype == 11)
(FEAT_FP16)

FMOV <Hd>, #<imm>

### Single-precision (ftype == 00)

FMOV <Sd>, #<imm>

### Double-precision (ftype == 01)

FMOV <Dd>, #<imm>

integer d = UInt(Rd);

integer datasize;

case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datasize = 16;
    else
      UNDEFINED;
  end

bits(datasize) imm = VFPExpandImm(imm8);

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<imm>` Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of `<imm>`, see Modified immediate constants in A64 floating-point instructions.

Operation

CheckFPAdvSIMDEnabled64();

V[d] = imm;
FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Feat_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | a | b | c | 1 | 1 | 1 | 1 | 1 | 1 | d | e | f | g | h | Rd |

FMOV <Vd>.<T>, #<imm>

if !HaveFP16Ext() then UNDEFINED;

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | O | p | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | a | b | c | 1 | 1 | 1 | 1 | 0 | 1 | d | e | f | g | h | Rd |

Single-precision (op == 0)

FMOV <Vd>.<T>, #<imm>

Double-precision (Q == 1 && op == 1)

FMOV <Vd>.2D, #<imm>

if cmode:op == '11111' then
  // FMOV Dn,#imm is in main FP instruction set
  if Q == '0' then UNDEFINED;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q";
For the single-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

<imm> is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

**Operation**

```
CheckFPAdvSIMDEnabled64();
V[rd] = imm;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ftype</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>Ra</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o1</td>
<td></td>
<td></td>
<td>o0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision (ftype == 11)

(FEAT_FP16)

FMSUB <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (ftype == 00)

FMSUB <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (ftype == 01)

FMSUB <Dd>, <Dn>, <Dm>, <Da>

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dr> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dr> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Dr> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Dr> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dr> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMUL (by element)**

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in **FPSCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 4 classes: **Scalar, half-precision**, **Scalar, single-precision and double-precision**, **Vector, half-precision** and **Vector, single-precision and double-precision**

**Scalar, half-precision**

**FEAT_FP16**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | L |   M | Rm | 1 | 0 | 0 | 1 | H | 0 | Rn | Rd |
| U  |    |    |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMUL <Hd>, <Hn>, <Vm>.H[index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

**Scalar, single-precision and double-precision**

**FEAT_FP32**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | L |   M | Rm | 1 | 0 | 0 | 1 | H | 0 | Rn | Rd |
| U  |    |    |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**FMUL <V<d>, <V<n>, <Vm>.<Ts>[index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
Vector, half-precision

(Feat_FP16)

|   | Q | O | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | L | M | Rm | 1 | 0 | 0 | 1 | H | 0 | Rn | Rd |
| U |   |   |   |   |   |   |   |   |   |   |   |   |     |   |   |   |   |   |   |   |   |   |

**FMUL** `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.H[<index>]`

if `!HaveFP16Ext()` then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Vector, single-precision and double-precision

|   | Q | O | 0 | 0 | 1 | 1 | 1 | 1 | 1 | sz | L | M | Rm | 1 | 0 | 0 | 1 | H | 0 | Rn | Rd |
| U |   |   |   |   |   |   |   |   |   |   |   |   |     |   |   |   |   |   |   |   |   |   |

**FMUL** `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<Ts>[<index>]`

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Assembler Symbols

-Hd- Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
-Hn- Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
-V- Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th></th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

-d- Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

-n- Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

-Vd- Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<vn>  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<vm>  
For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the “Rm” field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<ts>  
Is an element size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index>  
For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M” fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(esize) element1;
bias(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging (fpcr);
bias(128) result = if merge then V[n] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMULX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMUL(element1, element2, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMUL (scalar)**

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

- **Half-precision (ftype == 11)**
  (FEAT_FP16)

  \[ \text{FMUL } <Hd>, <Hn>, <Hm> \]

- **Single-precision (ftype == 00)**

  \[ \text{FMUL } <Sd>, <Sn>, <Sm> \]

- **Double-precision (ftype == 01)**

  \[ \text{FMUL } <Dd>, <Dn>, <Dm> \]

  integer \( d = \text{UInt}(Rd) \);
  integer \( n = \text{UInt}(Rn) \);
  integer \( m = \text{UInt}(Rm) \);

  integer \( esize; \)
  case \( ftype \) of
    when '00' \( esize = 32 \);
    when '01' \( esize = 64 \);
    when '10' UNDEFINED;
    when '11'
      if \( \text{HaveFP16Ext}() \) then
        \( esize = 16 \);
      else
        UNDEFINED;
  endcase

**Assembler Symbols**

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Hm>** Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Sm>** Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(\textit{esize}) operand1 = V[n];}
\texttt{bits(\textit{esize}) operand2 = V[m];}
\texttt{FPCRType fpcr = FPCR[];}
\texttt{boolean merge = \textit{IsMerging}(fpcr);}
\texttt{bits(128) result = if merge then V[n] else \textit{Zeros}();}
\texttt{bits(\textit{esize}) product = \textit{FPMul}(operand1, operand2, fpcr);}
\texttt{\textit{Elem}[result, 0, \textit{esize}] = product;}
\texttt{V[d] = result;}

Internal version only: isa v33.16decvel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FeaT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
V[d] = result;
```

**FMULX**

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision (FEAT_FP16)

![Scalar half precision encoding](image)

- if !HaveFP16Ext() then UNDEFINED;
  - integer d = UInt(Rd);
  - integer n = UInt(Rn);
  - integer m = UInt(Rm);
  - integer esize = 16;
  - integer datasize = esize;
  - integer elements = 1;

### Scalar single-precision and double-precision

![Scalar single-precision and double-precision encoding](image)

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- integer esize = 32 << UInt(sz);
- integer datasize = esize;
- integer elements = 1;

### Vector half precision (FEAT_FP16)

![Vector half precision encoding](image)
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Vector single-precision and double-precision

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(usize) element1;
bits(usize) element2;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
V[d] = result;
**FMULX (by element)**

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar, half-precision**, **Scalar, single-precision and double-precision**, **Vector, half-precision** and **Vector, single-precision and double-precision**

**Scalar, half-precision**

(Feat_fp16)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U

FMULX <Hd>, <Hn>, <Vm>.H[index]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

**Scalar, single-precision and double-precision**

(Feat_fp16)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U

FMULX <V>d>, <V>n>, <Vm>.<Ts>[index]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
Vector, half-precision
(FEAT_FP16)

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Vector, single-precision and double-precision

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12
Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rn  | Rd  |

| opc |

Half-precision (ftype == 11) (FEAT_FP16)

FNEG <Hd>, <Hn>

Single-precision (ftype == 00)

FNEG <Sd>, <Sn>

Double-precision (ftype == 01)

FNEG <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
   when '00'   esize = 32;
   when '01'   esize = 64;
   when '10'   UNDEFINED;
   when '11'   if HaveFP16Ext() then
e           esize = 16;
   else
         UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];

Elem[result, 0, esize] = FPNeg(operand);
V[d] = result;
FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**
(FEAT_FP16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
</tr>
</tbody>
</table>

FNEG <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
</tr>
</tbody>
</table>

FNEG <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}

**Operation**

```c
CheckFPAdvSIMDEnabled64();
b bits(
_datasetsize) operand = \text{V}[n];
b bits(
_datasetsize) result;
b bits(
esize) element;

\text{for } e = 0 \text{ to } \text{elements}-1
\quad \text{element} = \text{Elem}[\text{operand}, e, \text{esize}];
\quad \text{if neg then}
\quad \quad \text{element} = \text{FPNeg}(\text{element});
\quad \text{else}
\quad \quad \text{element} = \text{FPAbs}(\text{element});
\quad \text{Elem}[\text{result}, e, \text{esize}] = \text{element};

\text{V}[d] = \text{result};
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Half-precision (ftype == 11)  
(Feat_FP16)

FNMADD <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (ftype == 00)

FNMADD <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (ftype == 01)

FNMADD <Dd>, <Dn>, <Dm>, <Da>

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>  Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>  Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>  Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>  Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>  Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>  Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand1 = FPNeg(operand1);
operand2 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operand1, operand2, fpcr);

V[d] = result;
```
Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (ftype == 11)
(FEAT_FP16)

```
FNMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```clojure
d = Uint(Rd);
a = Uint(Ra);
n = Uint(Rn);
m = Uint(Rm);

esize = 32;
when '00' esize = 64;
when '10' UNDEFINED;
when '11'
if HaveFP16Ext() then
  esize = 16;
else
  UNDEFINED;
```

Assembler Symbols

- `<Dd>`: Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>`: Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>`: Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>`: Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>`: Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand = FPNeg(operand);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 1   | 1   | 1   | 0   | ftype | 1   | Rm  | 1   | 0   | 0   | 0   | 1   | 0   | op  |
```

**Half-precision (ftype == 11)**

(FEAT_FP16)

FNMUL <Hd>, <Hn>, <Hm>

**Single-precision (ftype == 00)**

FNMUL <Sd>, <Sn>, <Sm>

**Double-precision (ftype == 01)**

FNMUL <Dd>, <Dn>, <Dm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;

case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypen fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
product = FPNeg(product);
Elem[result, 0, esize] = product;

V[d] = result;
FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|-------------------|
| 0 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | Rd |

FRECPE <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|-------------------|
| 0 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rd |

FRECPE <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|-------------------|
| 0 | Q | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rd |

FRECPE <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|
| O  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  |

Rn | Rd

FRECPE \(<V_d>,<T>,<V_n>.<T>\)

integer \(d = \text{UINT}(R_d)\);
integer \(n = \text{UINT}(R_n)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer \(esize = 32 << \text{UINT}(sz)\);
integer \(datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64\);
integer \(\text{elements} = \frac{datasize}{esize}\);

Assembler Symbols

<\(H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\(H_n>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\(V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<\(d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<\(n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<\(V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\(T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "\(sz:Q\)":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\(V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\(\text{CheckFPAdvSIMDEnabled64}()\);
bits(datasize) operand = \(V[n]\);

\(\text{FPCRType} fpcr = \text{FPCR}[];\)
boolean \(\text{merge} = \text{elements} == 1 \text{ && IsMerging}(fpcr);\)
bits(128) result = if \(\text{merge}\) then \(V[d]\) else \(\text{Zeros}()\);
bits(esize) element;
for \(e = 0\) to \(\text{elements}-1\)
  element = \(\text{Elem}[(\text{operand}, e, esize)];\)
  \(\text{Elem}[(\text{result}, e, esize)] = \text{FPREcipEstimate}(element, fpcr[]);\)
\(V[d] = \text{result};\)
FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

Scalar half precision
(FEAT_FP16)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm  | 0  | 0  | 1  | 1  | 1  | 1  | Rn  | 0  | Rd |
```

FRECPS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm  | 1  | 1  | 1  | 1  | 1  | 1  | Rn  | 0  | Rd |
```

FRECPS <V>d>, <V>n>, <V>m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(FEAT_FP16)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm  | 0  | 0  | 1  | 1  | 1  | 1  | Rn  | 0  | Rd |
```

FRECPS <V>d>,<T>, <V>n><T>, <V>m><T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 1  | 1  | Rn | 1  | 1  | 1  | 1  | Rd |

**Assembler Symbols**

- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>`: Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<V>`: Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>`: Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>`: Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
V[d] = result;
Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for the source SIMD&FP register and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision  
**(FEAT_FP16)**

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 1   | 0   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 0   | Rn  | Rd  |

**FRECPX** <Hd>, <Hn>

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer esize = 16;

### Single-precision and double-precision

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 0   | Rn  | Rd  |

**FRECPX** <V><d>, <V><n>

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);
integer esize = 32 << **UInt**(sz);

### Assembler Symbols

- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<V>** Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- **<n>** Is the number of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(esize) operand = V[n];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

Elem[result, 0, esize] = FPREcpX(operand, fpcr);
V[d] = result;
```
**FRINT32X (scalar)**

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Floating-point

(Feat_FRINTTS)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |

<table>
<thead>
<tr>
<th>ftype</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

**Single-precision (ftype == 00)**

FRINT32X <Sd>, <Sn>

**Double-precision (ftype == 01)**

FRINT32X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);

### Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 32);

V[d] = result;
FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(Feat_FrIntTS)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| R | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | Rn | Rd |
| U | op |

FRINT32X <Vd>,<T>, <Vn>..<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;
FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the (corresponding) input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns (for the corresponding result value) the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FAET_FRINTTS)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1 1 0 0 x 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>ftype</td>
</tr>
</tbody>
</table>

Single-precision (ftype == 00)

FRINT32Z <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT32Z <Dd>, <Dn>

Assembler Symbols

<DD>  Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn>  Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd>  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 32);

V[d] = result;
FRINT32Z (vector)

Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(_FEAT_FRINTTS)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | Rn | Rd |

| 0 | 0 | 0 | 0 | 25 |
| 0 | 1 | 45 |
| 1 | 0 | RESERVED |
| 1 | 1 | 20 |

FRINT32Z <Vd>,<T>, <Vn>..<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem(operand, e, esize);
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;
FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FTAT_FRINTTS)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | x | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Rn | Rd |

```
ftype op
```

Single-precision (ftype == 00)

FRINT64X <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT64X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;Dd&gt;</th>
<th>&lt;Dn&gt;</th>
<th>&lt;Sd&gt;</th>
<th>&lt;Sn&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the 64-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
<td>Is the 64-bit name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
<td>Is the 32-bit name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
<td>Is the 32-bit name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypen fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then \texttt{V}[d] else \texttt{Zeros}();
bits(esize) operand = \texttt{V}[n];

\texttt{Elem}[\texttt{result}, 0, esize] = FP\texttt{RoundIntN}(\texttt{operand}, fpcr, \texttt{rounding}, 64);

\texttt{V}[d] = \texttt{result};
FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(Feat_FRINTTS)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| U | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FRINT64X <Vd>,<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem(operand, e, esize);
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;
FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the (corresponding) input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns (for the corresponding result value) the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FEAT_FRINTTS)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ftype</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single-precision (ftype == 00)
FRINT64Z <Sd>, <Sn>

Double-precision (ftype == 01)
FRINT64Z <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 64);
V[d] = result;
FRINT64Z (vector)

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_FRINTTS)

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Rn  Rd

U  op

FRINT64Z $<Vd>,<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[0]);

Assembler Symbols

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem(operand, e, esize);
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;
FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in\textit{FPCR}, the exception results in either a flag being set in\textit{FPSR}, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the\textit{CPACR_EL1}, \textit{CPTR_EL2}, and \textit{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |  |

rmode

Half-precision (ftype == 11) \hfill (FEAT\_FP16)

\texttt{FRINTA <Hd>, <Hn>}

Single-precision (ftype == 00)

\texttt{FRINTA <Sd>, <Sn>}

Double-precision (ftype == 01)

\texttt{FRINTA <Dd>, <Dn>}

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

- \texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Dn>} Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Hn>} Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Sd>} Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Sn>} Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypen fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bites(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, FPRounding_TIEAWAY, FALSE);
V[d] = result;
FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_FP16)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
0 | 2 | 0 | 1
```

FRINTA <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
0 | 2 | 0 | 1
```

FRINTA (vector)
integer d = \text{UInt}(\text{Rd});
integer n = \text{UInt}(\text{Rn});

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << \text{UInt}(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \div esize;

boolean exact = FALSE;
\text{FPRAunding} \text{rounding};
case U:01:o2 of
  when '0xx' rounding = \text{FPDecodeRounding}(o1:o2);
  when '100' rounding = \text{FPRoundingTIEAWAY};
  when '101' UNDEFINED;
  when '110' rounding = \text{FPRoundingMode}(\text{FPCR[]});
  when '111' rounding = \text{FPRoundingMode}(\text{FPCR[]});

\textbf{Assembler Symbols}

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>} For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>\texttt{Q}</th>
<th>\texttt{T}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>\texttt{sz}</th>
<th>\texttt{Q}</th>
<th>\texttt{T}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\textbf{Operation}

\texttt{CheckFPAdvSIMDEnabled64();}
bits(datasize) operand = \texttt{V}[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = \texttt{Elem}[operand, e, esize];
  \texttt{Elem}[result, e, esize] = \texttt{FPRoundInt}(element, \text{FPCR[]}, \text{rounding}, \text{exact});
\texttt{V}[d] = result;
FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (ftype == 11) (FEAT_FP16)

FRINTI <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTI <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTI <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);

**Assembler Symbols**

- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- <Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
**FRINTI (vector)**

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the **FPCR**, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision (**FEAT_FP16**)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**FRINTI <Vd>.<T>, <Vn>.<T>**

if !**HaveFP16Ext**() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

type integer esize = 16;
type integer datasize = if Q == '1' then 128 else 64;
type integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;

case U:o1:o2 of
when '0xx' rounding = FPDecodeRounding(o1:o2);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
when '111' rounding = FPRoundingMode(FPCR[]);

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | Rn | | Rd |
FRINTI \(<V_d><T>, <V_n><T>\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{ UInt}(Rn)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer \(esize = 32 << \text{UInt}(sz)\);
integer \(datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64;\)
integer \(elements = datasize \div esize\);

boolean exact = FALSE;
FP\(\text{Rounding}\) \(\text{rounding}\);
 caso \(U:o1:o2\) de
 cuando '0xx' \(\text{rounding} = \text{FP\text{Decode\text{Rounding}}}(o1:o2;\)
 cuando '100' \(\text{rounding} = \text{FP\text{Rounding\_TIEAWAY}};\)
 cuando '101' \(\text{UNDEFINED};\)
 cuando '110' \(\text{rounding} = \text{FP\text{Rounding\_Mode}}(\text{FPCR}[ ]);\) exact = TRUE;
 cuando '111' \(\text{rounding} = \text{FP\text{Rounding\_Mode}}(\text{FPCR}[ ]);\)

Assemblers Symbols

\(<V_d>\) Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>(sz) (&lt;T&gt;)</th>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the SIMD\&FP source register, encoded in the "Rn" field.

Operation

\text{CheckFPAdvSIMDEnabled64}();
bits(datasize) \(\text{operand} = V[n]\);
bits(datasize) \(\text{result}\);
bits(esize) \(\text{element}\);

for \(e = 0\) to \(\text{elements-1}\)
\(\text{element} = \text{Elem} [\text{operand}, e, esize];\)
\(\text{Elem} [\text{result}, e, esize] = \text{FPRound\text{Int}}(\text{element}, \text{FPCR}[ ], \text{rounding}, \text{exact});\)

\(V[d] = \text{result};\)
FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |

mode

Half-precision (ftype == 11)
(FEAT_FP16)

FRINTM <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTM <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTM <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPDecodeRounding('10');

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();

FPCRTypex fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in \texttt{FPCR}, the exception results in either a flag being set in \texttt{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Half-precision} and \texttt{Single-precision and double-precision}

### Half-precision (\texttt{FEAT\_FP16})

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| U | 02 | o1 |

\texttt{FRINTM <Vd>.<T>, <Vn>.<T>}

if !\texttt{HaveFP16Ext}() then UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
\texttt{FPRounding} rounding;
\texttt{case U:o1:o2 of}
  when '0xx' rounding = \texttt{FPDecodeRounding}(o1:o2);
  when '100' rounding = \texttt{FPRounding}\_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = \texttt{FPRoundingMode}(FPCR[{}]; exact = TRUE;
  when '111' rounding = \texttt{FPRoundingMode}(FPCR[{}];

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| U | 02 | o1 |
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;
FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0   0   0   1   1   1   1   0 | ftype 1          | 0   0   1   0   0   0   1   0   0   0   0 | Rn               | Rd               |

Half-precision (ftype == 11) (FEAT_FP16)

FRINTN <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTN <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTN <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
    esize = 16;
  else
    UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('00');

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\textbf{CheckFPAdvSIMDEnabled64}();

\textbf{FPCRTyp}e \ fpcr = FPCR[];
boolean merge = \textbf{IsMerging}(fpcr);
bits(128) result = if merge then \textbf{V}[d] else \textbf{Zeros}();
bits(esize) operand = \textbf{V}[n];

\textbf{Elem}[result, 0, esize] = \textbf{FPRoundInt}(operand, fpcr, rounding, FALSE);

\textbf{V}[d] = result;
FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPSCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision (FEAT_FP16)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 Rn Rd

FRINTN <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[0]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[0]);

Single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 1 1 0 0 1 0 Rn Rd

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;
FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (ftype == 11)
(Feat_FP16)

FRINTP <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTP <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTP <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
  FP_Rounding rounding;
  rounding = FPDecodeRounding('01');

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAddrSIMDEnabled64();

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
( FEAT_FP16 )

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[
\begin{array}{ll}
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & Rn & Rd
\end{array}
\]

FRINTP <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[
\begin{array}{ll}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & Rn & Rd
\end{array}
\]
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;
FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 0 ftype 1 0 0 1 1 1 0 1 0 0 0 0  Rn  Rd
```

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 0 0 1 1 1 0 ftype 1 0 0 1 1 1 0 1 0 0 0 0  Rn  Rd
```

Half-precision (ftype == 11) (FEAT_FP16)

FRINTX <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTX <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTX <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11'
  if HaveFP16Ext() then
    esize = 16;
  else
    UNDEFINED;
FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FProundInt(operand, fpcr, rounding, TRUE);

V[d] = result;
**FRINTX (vector)**

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the **FPCR**, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

### Half-precision

(Feat_FP16)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>
```

**FRINTX <Vd>.<T>, <Vn>.<T>**

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;

FPRounding rounding;

```
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

### Single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>
```
integer \( d = \text{UInt}(Rd); \) 
integer \( n = \text{UInt}(Rn);\)

if \( \text{sz:Q} == '10' \) then UNDEFINED;
integer esize = 32 \( \ll \) \( \text{UInt}(\text{sz}); \)
integer datasize = if \( Q == '1' \) then 128 else 64;
integer elements = datasize \( \div \) esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

**Assembler Symbols**

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\(\text{CheckFPAdvSIMDEnabled64}();\)

bits(datasize) operand = \( V[n]; \)
bits(datasize) result;
bits(esize) element;

for \( e = 0 \) to elements-1
  \( \text{element} = \text{Elem}[\text{operand}, e, \text{esize}]; \)
  \( \text{Elem}[\text{result}, e, \text{esize}] = \text{FPRoundInt}(\text{element}, \text{FPCR[]}, \text{rounding}, \text{exact}); \)

\( V[d] = \text{result}; \)
**FRINTZ (scalar)**

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see [*Floating-point exception traps*](#).

Depending on the settings in the *CPACR_EL1, CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Instruction Format](#)

| Bit 31 | Bit 30 | Bit 29 | Bit 28 | Bit 27 | Bit 26 | Bit 25 | Bit 24 | Bit 23 | Bit 22 | Bit 21 | Bit 20 | Bit 19 | Bit 18 | Bit 17 | Bit 16 | Bit 15 | Bit 14 | Bit 13 | Bit 12 | Bit 11 | Bit 10 | Bit 9 | Bit 8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 0      | 0      | 0      | 1      | 1      | 1      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 0      | 0      | Rn     | Rd     |

**Half-precision (ftype == 11)**

*(FEAT_FP16)*

FRINTZ <Hd>, <Hn>

**Single-precision (ftype == 00)**

FRINTZ <Sd>, <Sn>

**Double-precision (ftype == 01)**

FRINTZ <Dd>, <Dn>

```python
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPDencodeRounding('11');
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRTypedefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|
| Q 0 0 1 1 1 0 | 1 1 1 1 0 0 1 1 0 | Rn | Rd |
| U o2 | o1 |

FRINTZ <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|
| Q 0 0 1 1 1 0 | 1 0 0 0 0 1 1 1 0 | Rn | Rd |
| U o2 | o1 |
FRINTZ \(<Vd>.<T>, <Vn>.<T>\)

integer \(d\) = \(\text{UInt}(Rd)\);
integer \(n\) = \(\text{UInt}(Rn)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer \(esize\) = 32 << \(\text{UInt}(sz)\);
integer \(datasize\) = if \(Q == '1'\) then 128 else 64;
integer \(elements\) = \(datasize \div esize\);

boolean \(exact\) = FALSE;
FPRounding \(\text{rounding}\);
\(U:o1:o2\) of
  when '0xx' \(\text{rounding} = \text{FPDecodeRounding}(o1:o2)\);
  when '100' \(\text{rounding} = \text{FPRounding_TIEAWAY}\);
  when '101' \(\text{UNDEFINED}\);
  when '110' \(\text{rounding} = \text{FPRoundingMode}(FPCR[]); \(exact = \text{TRUE}\);
  when '111' \(\text{rounding} = \text{FPRoundingMode}(FPCR[])\);

**Assembler Symbols**

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\(\text{CheckFPAdvSIMDEnabled64}()\);
bits\((datasize)\) operand = \(V[n]\);
bits\((datasize)\) result;
bits\((esize)\) element;

for \(e = 0\) to \(elements - 1\)
  element = \(\text{Elem}[\text{operand, } e, \text{esize}]\);
  \(\text{Elem}[\text{result, } e, \text{esize}] = \text{FPRoundInt}(element, FPCR[], \text{rounding}, \text{exact})\);

\(V[d] = \text{result};\)
**FRSQRTE**

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR** or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

---

**Scalar half precision**

**(FEAT_FP16)**

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 0 | 1 1 1 1 0 0 1 1 0 1 1 0 | \textit{Rn} \textit{Rd}
```

**FRSQRTE** <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

---

**Scalar single-precision and double-precision**

**(FEAT_FP16)**

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 0 | 1 0 0 0 0 1 1 1 0 1 1 0 | \textit{Rn} \textit{Rd}
```

**FRSQRTE** <V><d>, <V><n>

integer d = UInt(Rd);
integer n =UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

---

**Vector half precision**

**(FEAT_FP16)**

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 \textit{Q} 1 0 1 1 1 1 0 | 1 1 1 1 1 0 0 1 1 1 0 1 1 0 | \textit{Rn} \textit{Rd}
```

**FRSQRTE** <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

\[
\begin{array}{cccccccccccccccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & 1 & sz & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & Rn & Rd
\end{array}
\]

**FRSQRTE** \(<Vd>.\langle T\rangle, \ <Vn>.\langle T\rangle\)**

integer \(d = \text{UInt}(Rd);\)
integer \(n = \text{UInt}(Rn);\)

if \(sz:Q == '10'\) then UNDEFINED;
integer esize = 32 << \text{UInt}(sz);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

- \(<Hd>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<Hn>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- \(<V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- \(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- \(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- \(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\[\text{CheckFPAdvSIMDEnabled64}();\]
\[\text{bits(datasize)} \text{ operand} = V[n];\]
\[\text{bits(esize)} \text{ element};\]
\[\text{FPCRType} \ fpcr = \text{FPCR}[];\]
\[\text{boolean merge} = \text{elements} == 1 \&\& \text{IsMerging}(fpcr);\]
\[\text{bits(128)} \text{ result} = \text{if merge then} \ V[d] \text{ else} \zeros();\]

for \(e = 0\) to \(\text{elements}-1\)
  \[\text{element} = \text{Elem}[\text{operand}, e, \text{esize}];\]
  \[\text{Elem}[\text{result}, e, \text{esize}] = \text{FPRSqrtEstimate}(\text{element}, fpcr);\]
\[V[d] = \text{result};\]
FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(_FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|
| 0 1 1 1 1 0 1 1 0 | Rm 0 0 1 1 1 1     | Rd                |

FRSQRTS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|
| 0 1 1 1 1 0 1 sz 1 | Rm 1 1 1 1 1 1     | Rd                |

FRSQRTS <V<d>, <V<n>, <V<m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(_FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|
| 0 Q 0 1 1 1 0 1 1 0 | Rm 0 0 1 1 1 1     | Rd                |

FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

FRSQRTS $<Vd>, <Vn>, <T>, <Vm>$

integer $d = \text{UInt}(Rd);$  
integer $n = \text{UInt}(Rn);$  
integer $m = \text{UInt}(Rm);$  
if sz:$Q == '10'$ then UNDEFINED;  
integer $e\text{size} = 32 << \text{UInt}(sz);$  
integer $d\text{ata}\text{size} = \text{if } Q == '1' \text{ then 128 else 64};$  
integer $e\text{lements} = d\text{ata}\text{size} \text{ DIV } e\text{size};$

Assembler Symbols

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Hn>$ Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Hm>$ Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, in the "Rd" field.

$<n>$ Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$<m>$ Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Vm>$ Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bhits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0   0   0   1   1   1   1   0</td>
<td>ftype</td>
</tr>
</tbody>
</table>

Half-precision (ftype == 11) (FEAT_FP16)

FSQRT <Hd>, <Hn>

Single-precision (ftype == 00)

FSQRT <Sd>, <Sn>

Double-precision (ftype == 01)

FSQRT <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;

case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];

Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FSQRT <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FSQRT <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FPSqrt(element, FPCR[]);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | Rm | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |

op

Half-precision (ftype == 11)
(Feat_FP16)

FSUB <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FSUB <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FSUB <Dd>, <Dn>, <Dm>

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRT pe fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPSub(operand1, operand2, fpcr);
V[d] = result;
FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPSCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Feat_FP16)

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 1 | 0 | 1 | Rn | Rd |
| U |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

**Single-precision and double-precision**

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
| U |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**INS (element)**

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `MOV (element)`.

---

**Asmmler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ts>` Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>xx100</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<index1>` Is the destination element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<index2>` Is the source element index encoded in “imm5:imm4”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm4&lt;3:0&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm4&lt;3:1&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3&gt;</td>
</tr>
</tbody>
</table>

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(128) result;

result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | imm5 | 0 | 0 | 0 | 1 | 1 | | Rn | Rd
```

**INS <Vd>.<Ts>[<index>], <R><n>**

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
```

**Assembler Symbols**

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ts>** Is an element size specifier, encoded in "imm5":

```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>
```

**<index>** Is the element index encoded in "imm5":

```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>
```

**<R>** Is the width specifier for the general-purpose source register, encoded in "imm5":

```
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>
```

**<n>** Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(128) result;

result = V[d];
Elem[result, index, esize] = element;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: No offset and Post-index.

No offset

```
0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | x | x | 1 | x | size | Rn | Rt
```
L

opcode

One register (opcode == 0111)

LD1 { <Vt>.<T> }, [<Xn|SP>]

Two registers (opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

Three registers (opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

Four registers (opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

```
0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | x | x | 1 | x | size | Rn | Rt
```
L

opcode
One register, immediate offset (Rm == 11111 && opcode == 0111)

LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>

One register, register offset (Rm != 11111 && opcode == 0111)

LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Two registers, register offset (Rm != 11111 && opcode == 1010)

LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Three registers, register offset (Rm != 11111 && opcode == 0110)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Four registers, register offset (Rm != 11111 && opcode == 0010)

LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

LD1 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

8-bit (opcode == 000)

LD1 { <Vt>.B }[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

LD1 { <Vt>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

LD1 { <Vt>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

LD1 { <Vt>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
8-bit, immediate offset (Rm == 11111 && opcode == 000)
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1

8-bit, register offset (Rm != 11111 && opcode == 000)
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared Decode

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);   // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);   // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);     // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);       // D[0-1]
    scale = 3;
  default
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
    integer datasize = if Q == '1' then 128 else 64;
    integer esize = 8 << scale;

Operation

if `HaveMTE2Ext()` then
    `SetTagCheckedInstruction`(`tag_checked`);

`CheckFPAdvSIMDEnabled64`();

bits(64) `address`;
bits(64) `offs`;
bits(128) `rval`;
bits(`esize`) `element`;
constant integer `ebytes` = `esize` DIV 8;

if `n` == 31 then
    `CheckSPAlignment`();
    `address` = `SP`[];
else
    `address` = `X`[`n`];

`offs` = `Zeros`();
if replicate then
    // load and replicate to all elements
    for `s` = 0 to `selem`-1
        `element` = `Mem`[`address`+`offs`, `ebytes`, `AccType_VEC`];
        // replicate to fill 128- or 64-bit register
        `V`[`t`] = `Replicate`(`element`, `datasize` DIV `esize`);
        `offs` = `offs` + `ebytes`;
        `t` = (`t` + 1) MOD 32;
else
    // load/store one element per register
    for `s` = 0 to `selem`-1
        `rval` = `V`[`t`];
        if `memop` == `MemOp_LOAD` then
            // insert into one lane of 128-bit register
            `Elem`[`rval`, `index`, `esize`] = `Mem`[`address`+`offs`, `ebytes`, `AccType_VEC`];
            `V`[`t`] = `rval`;
        else if `memop` == `MemOp_STORE`
            // extract from one lane of 128-bit register
            `Mem`[`address`+`offs`, `ebytes`, `AccType_VEC`] = `Elem`[`rval`, `index`, `esize`];
            `offs` = `offs` + `ebytes`;
            `t` = (`t` + 1) MOD 32;
if `wback` then
    if `m` != 31 then
        `offs` = `X`[`m`];
    if `n` == 31 then
        `SP`[] = `address` + `offs`;
    else
        `X`[`n`] = `address` + `offs`;

Operational information

If `PSTATE.DIT` is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | size | Rn | Rt |

LD1R { <Vt>,<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | size | Rm | 1  | 1  | 0  | 0  | size | Rn | Rt |

Immediate offset (Rm == 11111)

LD1R { <Vt>,<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD1R { <Vt>,<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#1</td>
</tr>
<tr>
<td>01</td>
<td>#2</td>
</tr>
<tr>
<td>10</td>
<td>#4</td>
</tr>
<tr>
<td>11</td>
<td>#8</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```c
integer init_scale = UInt(opcode<2:1>);
in{er scale = init_scale;
neger selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
in{er index;

case scale of
   when 3
      // load and replicate
      if L == '0' || S == '1' then UNDEFINED;
      scale = UInt(size);
      replicate = TRUE;
   when 0
      index = UInt(Q:S:size); // B[0-15]
   when 1
      if size<0> == '1' then UNDEFINED;
      index = UInt(Q:S:size<1>); // H[0-7]
   when 2
      if size<l> == '1' then UNDEFINED;
      if size<0> == '0' then
         index = UInt(Q:S); // S[0-3]
      else
         if S == '1' then UNDEFINED;
         index = UInt(Q); // D[0-1]
      scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
in{er datasize = if Q == '1' then 128 else 64;
in{er esize = 8 << scale;
```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 size Rn Rt</td>
</tr>
</tbody>
</table>

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 0 0 0 1 1 0 Rm 1 0 0 0 size Rn Rt</td>
</tr>
</tbody>
</table>

Immediate offset (Rm == 11111)

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD2 (single structure)**

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**8-bit (opcode == 000)**

LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]

**16-bit (opcode == 010 & size == x0)**

LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]

**32-bit (opcode == 100 & size == 00)**

LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]

**64-bit (opcode == 100 & S == 0 & size == 01)**

LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]

integer \( t = \text{UInt}(Rt); \)

integer \( n = \text{UInt}(Rn); \)

integer \( m = \text{integer\;UNKNOWN}; \)

boolean \( \text{wback} = \text{FALSE}; \)

boolean \( \text{tag\_checked} = \text{wback} || n != 31; \)

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**8-bit (opcode == 000)**

LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]

**16-bit (opcode == 010 & size == x0)**

LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]

**32-bit (opcode == 100 & size == 00)**

LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]

**64-bit (opcode == 100 & S == 0 & size == 01)**

LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]

integer \( t = \text{ UInt}(Rt); \)

integer \( n = \text{ UInt}(Rn); \)

integer \( m = \text{ integer\;UNKNOWN}; \)

boolean \( \text{ wback} = \text{ FALSE}; \)

boolean \( \text{ tag\_checked} = \text{ wback} || n != 31; \)
8-bit, immediate offset (Rm == 11111 && opcode == 000)
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
:index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
          For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
          For the 32-bit variant: is the element index, encoded in "Q:S".
          For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
offs = Zeros();
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
      offs = offs + ebytes;
      t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD2R**

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

**No offset**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------|-----------|
| 0   0  1  1  1  0  1  0  1  0  0  0  0  0  1  1  0  0 | size      |
| L   R | opcode | S         |
```

LD2R { <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>]

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------|-----------|
| 0   0  1  1  1  1  1 1 0 1 1 0 0 0 0 0 1 1 0 0 | size      |
| L   R | opcode | S         |
```

Immediate offset (Rm == 11111)

LD2R { <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD2R { <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>], <Xm>

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

**Assembler Symbols**

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
```

- `<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#2</td>
</tr>
<tr>
<td>01</td>
<td>#4</td>
</tr>
<tr>
<td>10</td>
<td>#8</td>
</tr>
<tr>
<td>11</td>
<td>#16</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

### Shared Decode

```plaintext
ingeneral integer init_scale = UInt(opcode<2:1>);
ingeneral integer scale = init_scale;
ingeneral integer selem = UInt(opcode<0>:R) + 1;
ingeneral boolean replicate = FALSE;
ingeneral integer index;

case scale of
  when 3 // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
      scale = 3;
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  integer datasize = if Q == '1' then 128 else 64;
  integer esize = 8 << scale;
```
Operation

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD3 (multiple structures)**

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:

![Diagram showing the operation of de-interleaving of a LD3.16 instruction]

A is a packed array of 3-element structures.
Each element is a 16-bit halfword.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |
```

<table>
<thead>
<tr>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>Rn</td>
</tr>
</tbody>
</table>

LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
ingter m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |
```

<table>
<thead>
<tr>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>Rn</td>
</tr>
</tbody>
</table>

**Immediate offset (Rm == 11111)**

LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

**Register offset (Rm != 11111)**

LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
ingter m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```
Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3 (single structure)

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| L  | R  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | x  | 1  | S  | size | Rn  | Rt  |

#### 8-bit (opcode == 001)

```
```

#### 16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

#### 32-bit (opcode == 101 && size == 00)

```
```

#### 64-bit (opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| L  | R  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | x  | x  | 1  | S  | size | Rn  | Rt  |

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```
8-bit, immediate offset (Rm == 11111 && opcode == 001)

LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)

LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt>   Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>  Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>  Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>   Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if \( \text{HaveMTE2Ext}() \) then
  \( \text{SetTagCheckedInstruction}(\text{tag\_checked}); \)

\( \text{CheckFPAdvSIMDEnabled64}() \);

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if \( n = 31 \) then
  \( \text{CheckSPAlignment}(); \)
  address = \( \text{SP}[] \);
else
  address = \( \text{X}[n] \);

offs = \( \text{Zeros}() \);
if replicate then
  // load and replicate to all elements
  for \( s = 0 \) to \( \text{selem} - 1 \)
    element = \( \text{Mem}[\text{address} + \text{offs}, \text{ebytes}, \text{AccType}_\text{VEC}] \);  
    // replicate to fill 128- or 64-bit register
    \( V[t] = \text{Replicate}(\text{element}, \text{datasize DIV esize}) \);
  offs = offs + ebytes;
  \( t = (t + 1) \text{ MOD 32} \);
else
  // load/store one element per register
  for \( s = 0 \) to \( \text{selem} - 1 \)
    rval = \( V[t] \);
    if memop == \text{MemOp\_LOAD} then
      // insert into one lane of 128-bit register
      \( \text{Elem}[\text{rval}, \text{index}, \text{esize}] = \text{Mem}[\text{address} + \text{offs}, \text{ebytes}, \text{AccType}_\text{VEC}] \);  
      \( V[t] = \text{rval} \);
    else // memop == \text{MemOp\_STORE}
      // extract from one lane of 128-bit register
      \( \text{Mem}[\text{address} + \text{offs}, \text{ebytes}, \text{AccType}_\text{VEC}] = \text{Elem}[\text{rval}, \text{index}, \text{esize}] \);
      offs = offs + ebytes;
    \( t = (t + 1) \text{ MOD 32} \);

if \( \text{wback} \) then
  if \( m \neq 31 \) then
    offs = \( \text{X}[m] \);
  if \( n = 31 \) then
    \( \text{SP}[] = \text{address} + \text{offs} \);
  else
    \( \text{X}[n] = \text{address} + \text{offs} \);

**Operational information**

If \( \text{PSTATE.DIT} = 1 \), the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0   0   1   1   1   0   1   0   0   0   0   0   1   1   1   0   size | Rn | Rt |
| L   R | opcode | S          |
```

LD3R \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>]

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0   0   1   1   1   0 | Rm | 1   1   1   0   size | Rn | Rt |
| L   R | opcode | S          |
```

Immediate offset (Rm == 11111)

```
LD3R \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD3R \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, [<Xn|SP>], <Xm>
```

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

**Assembler Symbols**

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
```

- `<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#3</td>
</tr>
<tr>
<td>01</td>
<td>#6</td>
</tr>
<tr>
<td>10</td>
<td>#12</td>
</tr>
<tr>
<td>11</td>
<td>#24</td>
</tr>
</tbody>
</table>

<Xm>  Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
ingtegr init_scale = UInt(opcode<2:1>);
ingtegr scale = init_scale;
ingtegr selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
ingtegr index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
    scale = 3;

    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
ingtegr datasize = if Q == '1' then 128 else 64;
ingtegr esize = 8 << scale;
```

```
Operation

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
offs = Zeros();
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t] = rval;
    else
      // memop == MemOp_STORE
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
    offs = offs + ebytes;
    t = (t + 1) MOD 32;

if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.
For an example of de-interleaving, see LD3 (multiple structures).
Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | size | Rn | Rt |

L

opcode


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | size | Rn | Rt |

L

opcode

Immediate offset (Rm == 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
      tt = (tt + 1) MOD 32;

if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | x | x | 1 | S | size | Rn | Rt |
| L | R | opcode |

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 & size == 0 & size == 01)


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 1 | 1 | 0 | 1 | 1 | L | R | x | x | 1 | S | size | Rn | Rt |
| opcode |

8-bit, immediate offset (Rm == 11111 && opcode == 001)


8-bit, register offset (Rm != 11111 && opcode == 001)


16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)


16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)


32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)


32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)


64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)


64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in “Q:S:size”.
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0:R> + 1);
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size); // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>); // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S); // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q); // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: No offset and Post-index.

No offset

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------------------------|----------------|----------------|
|       0 |   0 |  1 |  1 |  0 |  1 |  0 |  1 |  0 |  0 |  0 |  0 |  1 |  1 |  1 |  0 | size | Rn | Rt |
|---------------------------------|----------------|----------------|
```

LD4R { <Vt>..<T>, <Vt2>..<T>, <Vt3>..<T>, <Vt4>..<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------------------------|----------------|----------------|
|       0 |   0 |  1 |  1 |  1 |  1 |  0 |  1 |  1 |  1 |  0 | size | Rn | Rt |
|---------------------------------|----------------|----------------|
```

Immediate offset (Rm == 11111)

```
LD4R { <Vt>..<T>, <Vt2>..<T>, <Vt3>..<T>, <Vt4>..<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4R { <Vt>..<T>, <Vt2>..<T>, <Vt3>..<T>, <Vt4>..<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

- `<Vt>` is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vt2>` is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Vt3>` is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#4</td>
</tr>
<tr>
<td>01</td>
<td>#8</td>
</tr>
<tr>
<td>10</td>
<td>#16</td>
</tr>
<tr>
<td>11</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```


Operation

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n != 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point Non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
opc | 1 0 1 1 0 0 0 1 | imm7 | Rt2 | Rn | Rt
L
```

32-bit (opc == 00)

LDNP <St1>, <St2>, [{Xn|SP}{, #<imm>}]  

64-bit (opc == 01)

LDNP <Dt1>, <Dt2>, [{Xn|SP}{, #<imm>}]  

128-bit (opc == 10)

LDNP <Qt1>, <Qt2>, [{Xn|SP}{, #<imm>}]  

// Empty.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP (SIMD&FP).

Assembler Symbols

- `<Dt1>` Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Dt2>` Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<Qt1>` Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Qt2>` Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<St1>` Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<St2>` Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
  
  For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
  
  For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datalsize = 8 << scale;
b fractional offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

CheckFPAdvSIMDEnabled64();
b fractional address;
b fractional datalsize) datal;
b fractional datalsize) datal2;
constant integer dbytes = datalsize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
else
    address = X[n];
address = address + offset;
datal = Mem[address, dbytes, AccType_VECSTREAM];
datal2 = Mem[address+bytes, dbytes, AccType_VECSTREAM];
if rt unknown then
    datal = bits(datalsize) UNKNOWN;
    datal2 = bits(datalsize) UNKNOWN;
    V[t] = datal;
    V[t2] = datal2;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| opc | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | imm7 | Rt2 | Rn | Rt |

32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| opc | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | imm7 | Rt2 | Rn | Rt |

32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| opc | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | imm7 | Rt2 | Rn | Rt |
32-bit (opc == 00)
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}] 

64-bit (opc == 01)
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}] 

128-bit (opc == 10)
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}] 

boolean wback = FALSE;
boolean postindex = FALSE;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Assembler Symbols
<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
b bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;  // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
data1 = Mem[address, dbytes, AccType_VEC];
data2 = Mem[address+dbytes, dbytes, AccType_VEC];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;
if wback then
    if postindex then
        address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

Post-index

<table>
<thead>
<tr>
<th>size</th>
<th>0 1 1 1</th>
<th>0 0</th>
<th>x 1</th>
<th>0</th>
<th>imm9</th>
<th>0 1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

8-bit (size == 00 && opc == 01)

LDR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = Uint(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th>size</th>
<th>1 1 1 1</th>
<th>0 0</th>
<th>x 1</th>
<th>0</th>
<th>imm9</th>
<th>1 1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

opc
8-bit (size == 00 && opc == 01)
LDR <Bt>, [<Xn|SP>, #<simm>]

16-bit (size == 01 && opc == 01)
LDR <Ht>, [<Xn|SP>, #<simm>]

32-bit (size == 10 && opc == 01)
LDR <St>, [<Xn|SP>, #<simm>]

64-bit (size == 11 && opc == 01)
LDR <Dt>, [<Xn|SP>, #<simm>]

128-bit (size == 00 && opc == 11)
LDR <Qt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

(Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
Operation

CheckFPAdvSIMDEnabled64();
bits(64) address;
bots(datasize) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t];
    Mem[address, datasize DIV 8, AccType_VEC] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, AccType_VEC];
    V[t] = data;
if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>opc</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit (opc == 00)
LDR <St>, <label>

64-bit (opc == 01)
LDR <Dt>, <label>

128-bit (opc == 10)
LDR <Qt>, <label>

integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
  when '00' size = 4;
  when '01' size = 8;
  when '10' size = 16;
  when '11' UNDEFINED;

offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<Dt> Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
  SetTagCheckedInstruction(FALSE);

CheckFPAdvSIMDEnabled64();

data = Mem[address, size, AccType_VEC];
V[t] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>1</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

8-bit (size == 00 && opc == 01 && option != 011)

LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]  

8-bit (size == 00 && opc == 01 && option == 011)

LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>{, <extend> {<amount>}}])  

32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>, (<Wm>|<Xm>{, <extend> {<amount>}}])  

64-bit (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>{, <extend> {<amount>}}])  

128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>{, <extend> {<amount>}}])

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#4</td>
</tr>
</tbody>
</table>

**Shared Decode**

```c
integer n = UInt(Rn);
n integer t = UInt(Rt);
n integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
n integer datasize = 8 << scale;
n boolean tag_checked = memop != MemOp_PREFETCH;
```

LDR (register, SIMD&FP)
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
   SetTagCheckedInstruction(tag_checked);
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];
address = address + offset;
case memop of
   when MemOp_STORE
      data = V[t];
      Mem[address, datasize DIV 8, AccType_VEC] = data;
   when MemOp_LOAD
      data = Mem[address, datasize DIV 8, AccType_VEC];
      V[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>1</th>
<th>0</th>
<th>imm9</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 01)

LDUR <Bt>, [<Xn|SP>{, #<simm>}]

16-bit (size == 01 && opc == 01)

LDUR <Ht>, [<Xn|SP>{, #<simm>}]

32-bit (size == 10 && opc == 01)

LDUR <St>, [<Xn|SP>{, #<simm>}]

64-bit (size == 11 && opc == 01)

LDUR <Dt>, [<Xn|SP>{, #<simm>}]

128-bit (size == 00 && opc == 11)

LDUR <Qt>, [<Xn|SP>{, #<simm>}]

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
< Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag Checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 | 0 0 | 0 0 | H | 0 | Rn | Rd
```

MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
```

```c
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;
```

```c
element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 1 | 0 | 1 | Rn | Rd
U
```

MLA <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”.

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdySIMDAdabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | O  | L  | M  | Rm | 0  | 1  | 0  | 0  | H  | 0  | Rn | Rd | 2  | 0  |

MLS <Vd>..<T>, <Vn>..<T>, <Vm>..<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

MLS (by element)
<index> Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
```

bits(datasize) operand1 = V[n];
bits(idxsdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0   | O   | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 |
| Rm  | 1   | 0 | 0 | 1 | 0 | 1 |
| Rn  | Rd  |
```

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register. This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.

\[
\begin{array}{|c|c|c|c|}
\hline
0 & 1 & 1 & 0 \hline
0 & 0 & 0 & 0 \hline
imm5 & 0 & imm4 & 1 \hline
Rn & Rd & \hline
\end{array}
\]

MOV \(<Vd>\cdot<Ts>[<index1>], <Vn>\cdot<Ts>[<index2>]\)

is equivalent to

INS \(<Vd>\cdot<Ts>[<index1>], <Vn>\cdot<Ts>[<index2>]\)

and is always the preferred disassembly.

Assembler Symbols

- \(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<Ts>\) Is an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<index1>\) Is the destination element index encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

- \(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- \(<index2>\) Is the source element index encoded in "imm5:imm4":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm4&lt;3:0&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm4&lt;3:1&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3:2&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm4&lt;3&gt;</td>
</tr>
</tbody>
</table>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.
Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (general). This means:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode for this instruction.

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

<table>
<thead>
<tr>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm5</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 1 1 0 0 0 0</td>
<td>imm5</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

MOV <Vd>.<Ts>[<index>], <R><n>
is equivalent to
INS <Vd>.<Ts>[<index>], <R><n>
and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ts> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of DUP (element). This means:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode for this instruction.

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  \______________________\______________________\ \______________________\______________________\ \______________________\______________________\______________________
  0 1 0 1 1 1 0 0 0 0 \ imm5 \ 0 0 0 0 0 1 \ Rn \ Rd
```

**MOV <V><d>, <Vn>.<T>[<index>]**

is equivalent to

**DUP <V><d>, <Vn>.<T>[<index>]**

and is always the preferred disassembly.

**Assembler Symbols**

- `<V>` Is the destination width specifier, encoded in “imm5”:
  - `imm5`: 
    - x0000: RESERVED
    - xxxx1: B
    - xxxx10: H
    - xx100: S
    - x1000: D

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>` Is the element width specifier, encoded in “imm5”:
  - `imm5`: 
    - x0000: RESERVED
    - xxxx1: B
    - xxxx10: H
    - xx100: S
    - x1000: D

- `<index>` Is the element index encoded in “imm5”:
  - `imm5`: 
    - x0000: RESERVED
    - xxxx1: \imm5<4:1>
    - xxxx10: \imm5<4:2>
    - xx100: \imm5<4:3>
    - x1000: \imm5<4>

**Operation**

The description of DUP (element) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode for this instruction.

32-bit (Q == 0 && imm5 == xx100)

MOV <Wd>, <Vn>.S[index]

is equivalent to

UMOV <Wd>, <Vn>.S[index]

and is always the preferred disassembly.

64-bit (Q == 1 && imm5 == x1000)

MOV <Xd>, <Vn>.D[index]

is equivalent to

UMOV <Xd>, <Vn>.D[index]

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> For the 32-bit variant: is the element index encoded in "imm5<4:3>".
         For the 64-bit variant: is the element index encoded in "imm5<4>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of \texttt{ORR (vector, register)}. This means:

\begin{itemize}
  \item The encodings in this description are named to match the encodings of \texttt{ORR (vector, register)}.
  \item The description of \texttt{ORR (vector, register)} gives the operational pseudocode for this instruction.
\end{itemize}

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | Rm | 0 | 0 | 1 | 1 | 1 | Rn | Rd |
size
\end{verbatim}

\textbf{MOV} \texttt{<Vd>.<T>, <Vn>.<T>}

is equivalent to

\textbf{ORR} \texttt{<Vd>.<T>, <Vn>.<T>, <Vn>.<T>}

and is the preferred disassembly when \texttt{Rm == Rn}.

\textbf{Assembler Symbols}

\texttt{<Vd>} \begin{itemize}
  \item Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\end{itemize}

\texttt{<T>} \begin{itemize}
  \item Is an arrangement specifier, encoded in "Q":
  \begin{verbatim}
  Q | <T> |
  0 | 8B |
  1 | 16B |
  \end{verbatim}
\end{itemize}

\texttt{<Vn>} \begin{itemize}
  \item Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\end{itemize}

\textbf{Operation}

The description of \texttt{ORR (vector, register)} gives the operational pseudocode for this instruction.

\textbf{Operational information}

If PSTATE.DIT is 1:

\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**MOVI**

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
+--------+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 31    | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| Q | op | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | a | b | c | cmode | 0 | 1 | d | e | f | g | h | Rd |
```

8-bit (op == 0 && cmode == 1110)

MOVI <Vd>.<T>, #<imm8>{, LSL #0}

16-bit shifted immediate (op == 0 && cmode == 10x0)

MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifted immediate (op == 0 && cmode == 0xx0)

MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}

32-bit shifting ones (op == 0 && cmode == 110x)

MOVI <Vd>.<T>, #<imm8>, MSL #<amount>

64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

MOVI <Dd>, #<imm>

64-bit vector (Q == 1 && op == 1 && cmode == 1110)

MOVI <Vd>.2D, #<imm>

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;

bits(datasize) imm;

bits(64) imm64;

ImmediateOp operation;

```plaintext
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111' operation = ImmediateOp_MOVI;
        // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDEExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```
**Assembler Symbols**

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**imm**> Is a 64-bit immediate ‘aaaaaabb...bc', encoded in "a:b:c:d:e:f:g:h".

<**T**> For the 8-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;<strong>T</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

For the 16-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;<strong>T</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;<strong>T</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<**imm8**> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<**amount**> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

<table>
<thead>
<tr>
<th>cmode&lt;1&gt;</th>
<th>&lt;<strong>amount</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;<strong>amount</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

<table>
<thead>
<tr>
<th>cmode&lt;0&gt;</th>
<th>&lt;<strong>amount</strong>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  o The values of the data supplied in any of its registers.
  o The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  o The values of the data supplied in any of its registers.
  o The values of the NZCV flags.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 | 1 | 1 | 1 | size L M | Rm | 1 | 0 | 0 | H | 0 | Rn | Rd
```

MUL <Vd>.<T>, <Vn>..<T>, <Vm>.<Ts>[<index>]

```
integer idxdszie = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size `<Ts>` is H.

- `<Ts>` Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<index>` Is the element index, encoded in "size:L:H:M":
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![ASM SYMBOLS TABLE]

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<AsmSymD> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<AsmSymT> Is an arrangement specifier, encoded in "size:Q":

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean poly = (U == '1');

Assembler Symbols

<AsmSymD> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<AsmSymM> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem(operand1, e, esize);
    element2 = Elem(operand2, e, esize);
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MVN**

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of **NOT**. This means:

- The encodings in this description are named to match the encodings of **NOT**.
- The description of **NOT** gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & Rn & & & & & Rd & Rn
\end{array}
\]

**MVN** `<Vd>`. `<T>`, `<Vn>`. `<T>`

is equivalent to

**NOT** `<Vd>`. `<T>`, `<Vn>`. `<T>`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

The description of **NOT** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**MVNI**

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | a | b | c | cmode | 0 | 1 | d | e | f | g | h | Rd |
| 0 | Q | O | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | a | b | c | cmode | 0 | 1 | d | e | f | g | h | Rd |

**16-bit shifted immediate (cmode == 10x0)**

MVNI <Vd>,<T>, #<imm8>{, LSL #<amount>}

**32-bit shifted immediate (cmode == 0xx0)**

MVNI <Vd>,<T>, #<imm8>{, LSL #<amount>}

**32-bit shifting ones (cmode == 110x)**

MVNI <Vd>,<T>, #<imm8>, MSL #<amount>

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

**ImmediateOp** operation;

```c
case cmode:op of
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11111'
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the 16-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- **<imm8>** Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".
For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

<table>
<thead>
<tr>
<th>cmode&lt;1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

<table>
<thead>
<tr>
<th>cmode&lt;0&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd];
        result = operand AND NOT(imm);

V[rd] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
 NEG <V><d>, <V><n>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector

```
 NEG <Vd>..<T>, <Vn>..<T>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler Symbols

```
<Assembler Symbols>

<table>
<thead>
<tr>
<th>&lt;Assembler Symbols&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;V&gt; Is a width specifier, encoded in “size”:</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>0x</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>&lt;d&gt;</td>
</tr>
<tr>
<td>&lt;n&gt;</td>
</tr>
<tr>
<td>&lt;Vd&gt;</td>
</tr>
<tr>
<td>&lt;T&gt;</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);

    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias \textit{MVN}.

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
\end{verbatim}

\textbf{Assembler Symbols}

- \texttt{<Vd>} is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<T>} is an arrangement specifier, encoded in "Q":
  \begin{table}[h]
  \centering
  \begin{tabular}{|c|c|}
  \hline
  Q & <T> \\
  \hline
  0 & 8B \\
  1 & 16B \\
  \hline
  \end{tabular}
  \end{table}
- \texttt{<Vn>} is the name of the SIMD&FP source register, encoded in the "Rn" field.

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = NOT(element);
V[d] = result;
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `integer datasize = if Q == '1' then 128 else 64;`

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 OR operand2;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit (cmode == 0x1x)

ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}

16-bit (cmode == 10x1)

ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
when '0xx00' operation = ImmediateOp_MOVI;
when '0xx10' operation = ImmediateOp_ORR;
when '10x00' operation = ImmediateOp_MOVI;
when '10x10' operation = ImmediateOp_ORR;
when '110x0' operation = ImmediateOp_MOVI;
when '1110x' operation = ImmediateOp_MOVI;
when '11110' operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

<table>
<thead>
<tr>
<th>cmode&lt;1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.
For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ORR (vector, register)**

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).

<table>
<thead>
<tr>
<th>size</th>
<th>ORR &lt;Vd&gt;.&lt;T&gt;, &lt;Vn&gt;.&lt;T&gt;, &lt;Vm&gt;.&lt;T&gt;</th>
</tr>
</thead>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "Q":

  \[
  \begin{array}{c|c}
  Q & <T> \\
  \hline
  0 & 8B \\
  1 & 16B \\
  \end{array}
  \]

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector)</td>
<td>Rm == Rn</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
result = operand1 OR operand2;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over \{0,1\}*. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 1 | 1 | 1 | Rn | Rd |
| U |

PMUL \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
if \(U == ‘1’ \&\& \text{size} != ‘00’ \) then UNDEFINED;
if \(\text{size} == ‘11’ \) then UNDEFINED;
integer \(esize = 8 << \text{UInt}(\text{size})\);
integer \(\text{datasize} = \) if \(Q == ‘1’ \) then 128 else 64;
integer \(\text{elements} = \text{datasize} \div \text{esize}\);

\begin{array}{|c|c|}
\hline
\text{size} & Q \\
\hline
00 & 0 \\
00 & 1 \\
01 & x \\
1x & x \\
\hline
\end{array}

\begin{itemize}
\item <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\item <T> Is an arrangement specifier, encoded in "size:Q".
\item <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\item <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\end{itemize}

\begin{verbatim}
\text{Operation}
CheckFPAdvSIMDEnabled64();
bits(\text{datasize}) \text{operand1} = \text{V}[n];
bits(\text{datasize}) \text{operand2} = \text{V}[m];
bits(\text{datasize}) \text{result};
bits(\text{esize}) \text{element1};
bits(\text{esize}) \text{element2};
bits(\text{esize}) \text{product};
for e = 0 to \text{elements}-1
    \text{element1} = \text{Elem}([\text{operand1}, e, \text{esize}]);\n    \text{element2} = \text{Elem}([\text{operand2}, e, \text{esize}]);\n    \text{if poly then}
        \text{product} = \text{PolynomialMult}(\text{element1}, \text{element2})<\text{esize}-1:0>;
    \text{else}
        \text{product} = (\text{UInt}(\text{element1})*\text{UInt}(\text{element2}))<\text{esize}-1:0>;
    \text{Elem}([\text{result}, e, \text{esize}] = \text{product};
\text{V}[d] = \text{result};
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PMULL, PMULL2

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see Polynomial arithmetic over \(\{0, 1\}\).

The PMULL instruction extracts each source vector from the lower half of each source register. The PMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PMULL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1Q</td>
</tr>
</tbody>
</table>

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier; encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>O</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\begin{align*}
\text{bits}(\text{datasize}) \text{ operand1} &= \text{Vpart}[n, \text{part}] ; \\
\text{bits}(\text{datasize}) \text{ operand2} &= \text{Vpart}[m, \text{part}] ; \\
\text{bits}(2*\text{datasize}) \text{ result} ; \\
\text{bits}(\text{esize}) \text{ element1} ; \\
\text{bits}(\text{esize}) \text{ element2} ; \\
\end{align*}
\]

\[
\begin{align*}
\text{for } e = 0 \text{ to elements-1} \\
\quad &\text{element1} = \text{Elem}[\text{operand1}, e, \text{esize}] ; \\
\quad &\text{element2} = \text{Elem}[\text{operand2}, e, \text{esize}] ; \\
\quad &\text{Elem}[\text{result}, e, 2*\text{esize}] = \text{PolynomialMult}(\text{element1}, \text{element2}) ; \\
\end{align*}
\]

\[
\text{V}[d] = \text{result} ;
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
+-----------------+--------+--------+--------+--------+--------+
|   31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0 |
|-----------------+--------+--------+--------+--------+--------|
| 0              | Q      | 1      | 0      | 1      | 1      |
| size           | 1      | Rm     | 0      | 1      | 0      |
|                |        |        | 0      | 0      | 0      |
| Rd             |        |        |        |        |        |
| U              |        |        |        |        |        |
| 01             |        |        |        |        |        |
```

RADDHN(2) <Vd>, <Tb>, <Vn>, <Ta>, <Vm>, <Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

**Assembler Symbols**

- 2: Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

- <Vd>: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- <Tb>: Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- <Vn>: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- <Ta>: Is an arrangement specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- <Vm>: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]
\[
\text{bits(2*dsize) operand1} = V[n];
\]
\[
\text{bits(2*dsize) operand2} = V[m];
\]
\[
\text{bits(dsize) result};
\]
\[
\text{integer round\_const} = \text{if round then } 1 \ll (\text{esize} - 1) \text{ else 0;}
\]
\[
\text{bits(2*esize) element1};
\]
\[
\text{bits(2*esize) element2};
\]
\[
\text{bits(2*esize) sum};
\]
\[
\text{for e = 0 to elements-1}
\]
\[
\text{element1} = \text{Elem[operand1, e, 2*esize]};
\]
\[
\text{element2} = \text{Elem[operand2, e, 2*esize]};
\]
\[
\text{if sub\_op then}
\]
\[
\text{sum} = \text{element1 - element2};
\]
\[
\text{else}
\]
\[
\text{sum} = \text{element1 + element2};
\]
\[
\text{sum} = \text{sum + round\_const};
\]
\[
\text{Elem[result, e, esize] = sum<2*esize-1:esize>};
\]
\[
Vpart[d, part] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when \texttt{FEAT\_SHA3} is implemented.

Advanced SIMD
\texttt{(FEAT\_SHA3)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  |

\texttt{Rm} | \texttt{Rn} | \texttt{Rd}

\texttt{RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D}

if !\texttt{HaveSHA3Ext()} then UNDEFINED;
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\texttt{AArch64.CheckFPAdvSIMDEnabled();}

\texttt{bits(128) Vm = V[m];}
\texttt{bits(128) Vn = V[n];}
\texttt{V[d] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

RBIT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;

Assembler Symbols

&lt;Vd&gt; Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

&lt;T&gt; Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

&lt;Vn&gt; Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  for i = 0 to esize-1
    rev<(esize-1)-i> = element<i>;
  Elem[result, e, esize] = rev;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**REV16 (vector)**

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Rd</th>
<th>Rn</th>
<th>size</th>
<th>op</th>
<th>Q</th>
</tr>
</thead>
</table>
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | integer d = Uint(Rd); integer n = Uint(Rn);

// size=esize: B(0), H(1), S(1), D(S)
integer esize = 8 << Uint(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
// 64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
// 32+B = 1, 32+H = 2, 32+S = X, 32+D = X
// 16+B = 2, 16+H = X, 16+S = X, 16+D = X
//  8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
// 64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
// 32+B = 2, 32+H = 1, 32+S = X, 32+D = X
// 16+B = 1, 16+H = X, 16+S = X, 16+D = X
//  8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if Uint(op) + Uint(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
  when x then UNDEFINED;
endcase;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
// size=esize: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;
// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
\end{verbatim}

\textbf{Assembler Symbols}

\textbf{<Vd>}.<T>, <Vn>.<T>

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;
integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
\end{verbatim}

\textbf{Assembler Symbols}

\textbf{<Vd>}: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textbf{<T>}: Is an arrangement specifier, encoded in “size:Q”:

\begin{verbatim}
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{<Vn>}: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{V}[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for \text{c} = 0 \text{ to containers-1}
    rev_element = element + \text{elements_per_container - 1};
    for \text{e} = 0 \text{ to elements_per_container-1}
        \text{Elem}[result, rev_element, esize] = \text{Elem}[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
\text{V}[d] = result;

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
// 64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
// 32+B = 1, 32+H = 2, 32+S = X, 32+D = X
// 16+B = 2, 16+H = X, 16+S = X, 16+D = X
// 8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
// 64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
// 32/B = 2, 32+H = 1, 32+S = X, 32+D = X
// 16/B = 1, 16+H = X, 16+S = X, 16+D = X
// 8/B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
endcase;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\texttt{CheckFPAdvSIMDEnabled64();}
\begin{align*}
\text{bits(datasize) operand} &= \text{V}[n]; \\
\text{bits(datasize) result;} \\
\text{integer element} &= 0; \\
\text{integer rev\_element;} \\
\text{for } c = 0 \text{ to containers - 1} \\
&\quad \text{rev\_element} = \text{element} + \text{elements\_per\_container} - 1; \\
&\quad \text{for } e = 0 \text{ to elements\_per\_container - 1} \\
&\quad \quad \text{Elem}[\text{result, rev\_element, esize}] = \text{Elem}[\text{operand, element, esize}]; \\
&\quad \quad \text{element} = \text{element} + 1; \\
&\quad \quad \text{rev\_element} = \text{rev\_element} - 1; \\
\text{V}[d] &= \text{result};
\end{align*}

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 0 1 1 1 0 != 0000 immh 1 0 0 0 1 1 Rn Rd

RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier; encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "immh":

RSHRN, RSHRN2
Is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

```
<shift>
```

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN.

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rd | 0 | 1 | 1 | 0 | 0 | Rn | Rd |
| 0 | U | 1 |

RSUBHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 <= UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;
for e = 0 to elements-1
  element1 = Elem[operand1, e, 2*esize];
  element2 = Elem[operand2, e, 2*esize];
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  sum = sum + round_const;
  Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>Size (Q)</th>
<th>Rm</th>
<th>Rn</th>
<th>Rd</th>
<th>Ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{SABA } <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(Rd); \\
\text{integer } n &= \text{UInt}(Rn); \\
\text{integer } m &= \text{UInt}(Rm); \\
\text{if } \text{size} == \text{'11'} \text{ then UNDEFINED;} \\
\text{integer } \text{esize} &= 8 <\text{ UInt(size);} \\
\text{integer } \text{datasize} &= \text{if } \text{Q} == \text{'1'} \text{ then } 128 \text{ else } 64; \\
\text{integer } \text{elements} &= \text{datasize} \text{ DIV } \text{esize}; \\
\text{boolean } \text{unsigned} &= (\text{U} == \text{'1'}); \\
\text{boolean } \text{accumulate} &= (\text{ac} == \text{'1'});
\end{align*}
\]

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register. The SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SABAL(2) <Vd>, <Ta>, <Vn>, <Vm>, <Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<p>| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------|-------------------|-------------------|</p>
<table>
<thead>
<tr>
<th>size</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>0 1 1</td>
<td>0 1 1 0</td>
</tr>
</tbody>
</table>
```

**SABD** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction extracts each source vector from the lower half of each source register, while the SABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>Q</th>
<th>size</th>
<th>Rmd</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
SABDL2(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

- integer \( d = \text{UInt}(Rd) \)
- integer \( n = \text{UInt}(Rn) \)
- integer \( m = \text{UInt}(Rm) \)

```plaintext
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>

Is an arrangement specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>

Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vm>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | 0 | 0 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Rd
U | 0 | 1 | op

SADALP <Vd>.<Ta>, <Vn>.<Tb>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

```
size  Q    <Ta>
00    0    4H
00    1    8H
01    0    2S
01    1    4S
10    0    1D
10    1    2D
11    x    RESERVED
```

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
size  Q    <Tb>
00    0    8B
00    1    16B
01    0    4H
01    1    8H
10    0    2S
10    1    4S
11    x    RESERVED
```
Operation

```c
CheckFPAdvSIMDEnabled64();
bites(datasize) operand = V[n];
bites(datasize) result;

bites(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register. The SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>**

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>**

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Tb>**

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Rd |
| U |

SADDLV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);
V[d] = sum<2*esize-1:0>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register. The SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb>  Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, 2*esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
### SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>rmode</th>
<th>opcode</th>
</tr>
</thead>
</table>

#### 32-bit to half-precision (sf == 0 & ftype == 11)

(FEAT_FP16)

SCVTF <Hd>, <Wn>, #<fbits>

#### 32-bit to single-precision (sf == 0 & ftype == 00)

SCVTF <Sd>, <Wn>, #<fbits>

#### 32-bit to double-precision (sf == 0 & ftype == 01)

SCVTF <Dd>, <Wn>, #<fbits>

#### 64-bit to half-precision (sf == 1 & ftype == 11)

(FEAT_FP16)

SCVTF <Hd>, <Xn>, #<fbits>

#### 64-bit to single-precision (sf == 1 & ftype == 00)

SCVTF <Sd>, <Xn>, #<fbits>

#### 64-bit to double-precision (sf == 1 & ftype == 01)

SCVTF <Dd>, <Xn>, #<fbits>

```plaintext
d = UInt(Rd);
n = UInt(Rn);

tsize = if sf == '1' then 64 else 32;
fltsize =
rounding = FPRounding(FPCR[]);
```

```plaintext
case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  if sf == '0' & scale<5> == '0' then UNDEFINED;
fracbits = 64 - UInt(scale);
```
Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bites(fsize) fltval;
bites(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding);
V[d] = fltval;

Internal version only: isa v33.16decrcl, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SCVTF (scalar, integer)**

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT_FP16)

SCVTF `<Hd>`, `<Wn>`

### 32-bit to single-precision (sf == 0 && ftype == 00)

SCVTF `<Sd>`, `<Wn>`

### 32-bit to double-precision (sf == 0 && ftype == 01)

SCVTF `<Dd>`, `<Wn>`

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT_FP16)

SCVTF `<Hd>`, `<Xn>`

### 64-bit to single-precision (sf == 1 && ftype == 00)

SCVTF `<Sd>`, `<Xn>`

### 64-bit to double-precision (sf == 1 && ftype == 01)

SCVTF `<Dd>`, `<Xn>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
   when '00'
      fltsize = 32;
   when '01'
      fltsize = 64;
   when '10'
      UNDEFINED;
   when '11'
      if HaveFP16Ext() then
         fltsize = 16;
      else
         UNDEFINED;
   rounding = FPRoundingMode(FPCR[]);
```
Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<HD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<SD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<XN> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<WN> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, FALSE, fpcr, rounding);
V[d] = fltval;
SCVTTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1 | 1 | 1 | 1 | 1 | 0 | != | 0000 | immh | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd |
| U  |  immh |
```

**SCVTTF <V><d>, <V><n>, #<fbits>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datashere = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);

### Vector

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0  | 0 | 1 | 1 | 1 | 1 | 1 | 0 | != | 0000 | immh | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd |
| U  | immh |
```

**SCVTTF <Vd>.<T>, <Vn>.<T>, #<fbits>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datashere = if Q == '1' then 128 else 64;
integer elements = datashere DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);

### Assembler Symbols

<V> Is a width specifier, encoded in “immh”:
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEEK Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x RESERVED</td>
</tr>
<tr>
<td>011x</td>
<td>0 4H</td>
</tr>
<tr>
<td>011x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEEK Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>011x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding);

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped. It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

Scalar half precision

(FEAT_FP16)

```
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
```

```
U
```

SCVTF <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

```
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
```

```
U
```

SCVTF <V<d>, <V<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector half precision

(FEAT_FP16)

```
| 0 | Q | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
```

```
U
```
SCVTFT <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 0  | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |

SCVTFT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

FPCRTypemfc = FPCR();
boolean merge = elements == 1 & IsMerging(mfc);
bits(128) result = if merge then V[d] else Zeros();

FPRounding rounding = FPRoundingMode(mfc);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, mfc, rounding);
V[d] = result;
SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

**Note**

`ID_AA64ISAR0_EL1.DP` indicates whether this instruction is supported.

**Vector (FEAT_DotProd)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 0  | 0  | 1  | 1  | L  | M  | Rm | 1  | 1  | 0  | H  | 0  | Rn | Rd |

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in "Q":
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in "Q":
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- `<index>` Is the element index, encoded in the "H:L" fields.
Operation

CheckFPAdvSIMDEnabled64();
because(datasize) operand1 = V[n];
because(128) operand2 = V[m];
because(datasize) result = V[d];

for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

$ID_AA64ISAR0_EL1.DP$ indicates whether this instruction is supported.

Vector (FEAT_DotProd)

$SDOT <Vd>..<Ta>., <Vn>..<Tb>, <Vm>..<Tb>$

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = $UInt$(Rd);
integer n = $UInt$(Rn);
integer m = $UInt$(Rm);
integer esize = 8 << $UInt$(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

$<Vd>$  Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

$<Ta>$  Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

$<Vn>$  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Tb>$  Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>1</td>
<td>168</td>
</tr>
</tbody>
</table>

$<Vm>$  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

`CheckFPAdvSIMDEnabled64();`

```
bounds(datasize) operand1 = V[n];
bounds(datasize) operand2 = V[m];
bounds(datasize) result;

result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
SHA1C

SHA1 hash update (choose).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SHA1C \(<Qd>, <Sn>, <Vm>\).4S

```bash
text
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

\(<Qd>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

\(<Sn>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];    // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;
for e = 0 to 3
    t = SHA1Choose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1 fixed rotate.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | Rn |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SHA1H <Sd>, <Sn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n];    // read element [0] only, [1-3] zeroed
V[d] = ROL(operand, 30);
```

Operational information

- If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
SHA1M

SHA1 hash update (majority).

|       | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Rm    | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rn    | 0  | 0  | 1  | 0  | 0  | 0  |
| Rd    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SHA1M \(<Q_d>, <S_n>, <V_m>.4S\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
if \(!\text{HaveSHA1Ext}()\) then UNDEFINED;

Assembler Symbols

\(<Q_d>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
\(<S_n>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(<V_m>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\(\text{AArch64.CheckFPAdvSIMDEnabled}()\);

\begin{align*}
\text{bits}(128) & \quad X = V[d]; \\
\text{bits}(32) & \quad Y = V[n]; \quad \text{// Note: 32 not 128 bits wide} \\
\text{bits}(128) & \quad W = V[m]; \\
\text{bits}(32) & \quad t; \\
\text{for } e = 0 \text{ to } 3 \\
\quad & \quad t = \text{SHA}majority(X<63:32>, X<95:64>, X<127:96>); \\
\quad & \quad Y = Y + \text{ROL}(X<31:0>, 5) + t + \text{Elem}[W, e, 32]; \\
\quad & \quad X<63:32> = \text{ROL}(X<63:32>, 30); \\
\quad & \quad <Y, X> = \text{ROL}(Y:X, 32); \\
\quad & \quad V[d] = X;
\end{align*}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16dec6el, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA1P

SHA1 hash update (parity).

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SHA1P <Qd>, <Sn>, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];  // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;
for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU0

SHA1 schedule update 0.

| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Rm | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rd |

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA1SU1

SHA1 schedule update 1.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SHA1SU1 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256H

SHA256 hash update (part 1).

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SHA256H \(<Qd>, <Qn>, <Vm>\).4S

integer \(d = \text{UInt}(Rd);\)
integer \(n = \text{UInt}(Rn);\)
integer \(m = \text{UInt}(Rm);\)
if \(!\text{HaveSHA256Ext}()\) then UNDEFINED;

Assembler Symbols

\(<Qd>\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
\(<Qn>\) Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(<Vm>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\(\text{AArch64.CheckFPAdvSIMDEnabled}();\)

bits(128) result;
result = \(\text{SHA256hash}(V[d], V[n], V[m], \text{TRUE});\)
\(V[d] = \text{result};\)

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA256H2

SHA256 hash update (part 2).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 1 1 1 0 0 0 | Rm | 0 1 0 1 0 0 | Rn | Rd |

SHA256H2, <Qd>, <Qn>, <Vm>.

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

- `<Qd>` is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- `<Qn>` is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256SU0

SHA256 schedule update 0.

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SHA256SU0  <Vd>.4S,  <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd>  Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>  Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;
for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256SU1

SHA256 schedule update 1.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 0 1 0 1 1 1 1 0 | 0 0 0 | 0 0 1 | 1 0 0 0 | Rm | 0 1 1 0 0 0 | Rn | Rd |

SHA256SU1  <Vd>.4S,  <Vn>.4S,  <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e-2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

Advanced SIMD (FEAT_SHA512)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 1 1 1 0 0 1 1 | Rm | 1 0 0 0 0 0 | Rn | Rd

SHA512H <Qd>, <Qn>, <Vm>.2D

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\[
\text{AArch64.CheckFPAdvSIMDEnabled();}
\]

\[
\text{bits(128) Vtmp;}
\]

\[
\text{bits(64) MSigma1;}
\]

\[
\text{bits(128) X = V[n];}
\]

\[
\text{bits(128) Y = V[m];}
\]

\[
\text{bits(128) W = V[d];}
\]

\[
\text{MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);}\]

\[
\text{Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);}\]

\[
\text{Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);}\]

\[
\text{tmp = Vtmp<127:64> + Y<63:0>;}\]

\[
\text{MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);}\]

\[
\text{Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);}\]

\[
\text{Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);}\]

\[
V[d] = Vtmp;\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA512* is implemented.

**Advanced SIMD**

(*FEAT_SHA512*)

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
</tr>
</tbody>
</table>
```

SHA512H2 <Qd>, <Qn>, <Vm>.2D

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

**Assembler Symbols**

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

NSigma0 = ROR(Y<63:0>, 28) EOR ROR(Y<63:0>, 34) EOR ROR(Y<63:0>, 39);
Vtmp<127:64> = (X<63:0> AND Y<127:64>) EOR (X<63:0> AND Y<63:0>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND Y<63:0>) EOR (Vtmp<127:64> AND Y<127:64>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + W<63:0>);

V[d] = Vtmp;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

**Advanced SIMD**
*(FEAT_SHA512)*

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

**SHA512SU0** \(<Vd>.2D, <Vn>.2D\)

```plaintext
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

**Assembler Symbols**

- \(<Vd>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- \(<Vn>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) W = V[d];
sig0 = ROR(W<127:64>, 1) EOR ROR(W<127:64>, 8) EOR ('0000000':W<127:71>);
Vtmp<63:0> = W<63:0> + sig0;
sig0 = ROR(X<63:0>, 1) EOR ROR(X<63:0>, 8) EOR ('0000000':X<63:7>);
Vtmp<127:64> = W<127:64> + sig0;
V[d] = Vtmp;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA512 is implemented.

Advanced SIMD
(_FEAT_SHA512)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D |

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];
sig1 = ROR(X<127:64>, 19) EOR ROR(X<127:64>, 61) EOR ('000000':X<127:70>);
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR ('000000':X<63:6>);
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ○ The values of the data supplied in any of its registers.
  ○ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ○ The values of the data supplied in any of its registers.
  ○ The values of the NZCV flags.
**SHADD**

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see **SRHADD**.

Depending on the settings in the **CPACR_EL1**, **CPTER_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |
| U |

**SHADD** \(<V_d>\).\(<T>\)\,, \(<V_n>\).\(<T>\)\,, \(<V_m>\).\(<T>\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

**Assembler Symbols**

\(<V_d>\)  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\)  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<V_n>\)  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\)  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

\[
\begin{array}{cccccccccccccccccccccc}
\hline
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \lnot & 0000 & \text{immb} & 0 & 1 & 0 & 1 & 0 & 1 & \text{Rn} & \text{Rd} & \text{immh}
\end{array}
\]

\[
\text{SHL } <V><d>, <V><n>, \#<shift>
\]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
if \( \text{immh}:3 != '1' \) then UNDEFINED;
integer esize = 8 \ll 3;
integer datasize = esize;
integer elements = 1;
integer shift = \text{UInt}(\text{immh}:(\text{immb}) - \text{esize});

Vector

\[
\begin{array}{cccccccccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \lnot & 0000 & \text{immb} & 0 & 1 & 0 & 1 & 0 & 1 & \text{Rn} & \text{Rd} & \text{immh}
\end{array}
\]

\[
\text{SHL } <Vd>\cdot<T>, <Vn>\cdot<T>, \#<shift>
\]

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
if \( \text{immh} == '0000' \) then \text{SEE(asimdimm)};
if \( \text{immh}:3:Q == '10' \) then UNDEFINED;
integer esize = 8 \ll \text{HighestSetBit}(\text{immh})
integer datasize = if \( Q == '1' \) then 128 else 64;
integer elements = datasize \div \text{esize};
integer shift = \text{UInt}(\text{immh}:(\text{immb}) - \text{esize});

Assembler Symbols

\(<V>\quad \text{Is a width specifier, encoded in “immh”:}\)

\[
\begin{array}{c|c}
\text{immh} & \text{<V>} \\
0\text{xxx} & \text{RESERVED} \\
1\text{xxx} & D
\end{array}
\]

\(<d>\quad \text{Is the number of the SIMD&FP destination register, in the “Rd” field.}\)

\(<n>\quad \text{Is the number of the first SIMD&FP source register, encoded in the ”Rn” field.}\)

\(<Vd>\quad \text{Is the name of the SIMD&FP destination register, encoded in the ”Rd” field.}\)

\(<T>\quad \text{Is an arrangement specifier, encoded in “immh:Q”:}\)
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>((\text{UInt}(\text{immh:immb})-64))</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>((\text{UInt}(\text{immh:immb})-8))</td>
</tr>
<tr>
<td>001x</td>
<td>((\text{UInt}(\text{immh:immb})-16))</td>
</tr>
<tr>
<td>01xx</td>
<td>((\text{UInt}(\text{immh:immb})-32))</td>
</tr>
<tr>
<td>1xxx</td>
<td>((\text{UInt}(\text{immh:immb})-64))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAAdvSIMDEnabled64();
bits(datasize) operand = \text{V}[n];
bits(datasize) result;
for \(e = 0\) to elements-1  
  \text{Elem}[result, e, esize] = LSL(\text{Elem}[operand, e, esize], shift);
\text{V}[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. The SHLL instruction extracts vector elements from the lower half of the source register. The SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

SHLL{2}  <Vd>, <Ta>, <Vn>.<Tb>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE;    // Or TRUE without change of functionality

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SHRN, SHRN2**

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see **RSHRN**.

The **RSHRN** instruction writes the vector to the lower half of the destination register and clears the upper half, while the **RSHRN2** instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>immh</th>
<th>Rd</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1 0 0 0 0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```
<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**SHSUB** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

**Assembler Symbols**

- **<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** is an arrangement specifier, encoded in "size:Q":
  
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
ingeneger element2;
ingeneger diff;
for e = 0 to elements-1
    element1 = Int(Elem(operand1, e, esize], unsigned);  
    element2 = Int(Elem(operand2, e, esize], unsigned);  
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.

\[
\begin{array}{c}
\text{Vn.B[7] before operation} \\
\text{Vd.B[7] after operation} \\
\text{Vd.B[7] after operation}
\end{array}
\]

Depending on the settings in the \texttt{CPACR_EL1, CPTR_EL2,} and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

\[
\begin{array}{c}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & !\text{=} 0000 & \text{immb} & 0 & 1 & 0 & 1 & 0 & 1 & \text{Rn} & \text{Rd}
\end{array}
\]

\[
\text{immmh}
\]

\[
\text{SLI} <V><d>, <V><n>, #<shift>
\]

\[
\begin{array}{c}
\text{integer } d = \text{UInt}(Rd); \\
\text{integer } n = \text{UInt}(Rn); \\
\text{if } \text{immmh<3> \! = \! '1'} \text{ then UNDEFINED}; \\
\text{integer esize} = 8 \ll 3; \\
\text{integer datasize} = \text{esize}; \\
\text{integer elements} = 1; \\
\text{integer shift} = \text{UInt}(\text{immmh:immb}) - \text{esize};
\end{array}
\]

**Vector**

\[
\begin{array}{c}
0 & Q & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & !\text{=} 0000 & \text{immb} & 0 & 1 & 0 & 1 & 0 & 1 & \text{Rn} & \text{Rd}
\end{array}
\]

\[
\text{immmh}
\]

\[
\text{SLI} <Vd>.<T>, <Vn>.<T>, #<shift>
\]

\[
\begin{array}{c}
\text{integer } d = \text{UInt}(Rd); \\
\text{integer } n = \text{UInt}(Rn); \\
\text{if } \text{immmh \! = \! '0000'} \text{ then } \text{SEE(asimdimm)}; \\
\text{if } \text{immmh<3>:Q \! = \! '10'} \text{ then UNDEFINED}; \\
\text{integer esize} = 8 \ll \text{HighestSetBit}(\text{immmh}); \\
\text{integer datasize} = \text{if } Q \text{ == '1' then 128 else 64}; \\
\text{integer elements} = \text{datasize DIV esize}; \\
\text{integer shift} = \text{UInt}(\text{immmh:immb}) - \text{esize};
\end{array}
\]
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>0xxx</th>
<th>1xxx</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n];
bias(datasize) operand2 = V[d];
bias(datasize) result;
bias(esize) mask = LSL(Ones(esize), shift);
bias(esize) shifted;
for e = 0 to elements-1
  shifted = LSL(Elem[operand, e, esize], shift);
  Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(FEAT_SM3)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL (Vm<127:96>, 15):ROL (Vm<95:64>, 15):ROL (Vm<63:32>, 15));

for i = 0 to 3
  if i == 3 then
    result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR ROL(result<(32*i)+31:(32*i)>, 15);
end if

V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.

Internal version only: isa v31.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information. This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(FEAT_SM3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | Rm |
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | Rd |

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

If !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD (FEAT_SM3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra | 0  | Ra |

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
bits(128) result;
result<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
result<95:0> = Zeros();
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when FEAT_SM3 is implemented.

**Advanced SIMD (FEAT_SM3)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S<imm2>**

```plaintext
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- `<imm2>` is a 32-bit element indexed out of `<Vm>`, encoded in "imm2".

**Operation**

```plaintext
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;
WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT1B

SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, \( V_d \), that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, \( V_n \), with a rotation left by \( 12 \) of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, \( V_m \).

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by \( 9 \).

This instruction is implemented only when \( \text{FEAT\_SM3} \) is implemented.

**Advanced SIMD (FEAT\_SM3)**

<table>
<thead>
<tr>
<th>( Rm )</th>
<th>( \text{imm2} )</th>
<th>( Rn )</th>
<th>( Rd )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 0 1 0</td>
<td>1 0</td>
<td>0 1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

SM3TT1B \(<V_d>.4S, <V_n>.4S, <V_m>.S[<\text{imm2}>]>)

if \(!\text{HaveSM3Ext}()\) then UNDEFINED;
integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);
integer \( i = \text{UInt}(\text{imm2}) \);

**Assembler Symbols**

- \(<V_d>\) Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- \(<V_n>\) Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- \(<V_m>\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- \(<\text{imm2}>\) Is a 32-bit element indexed out of \(<V_m>\), encoded in "imm2".

**Operation**

```assembly
AArch64.CheckFPAdvSIMDEnabled();

\begin{align*}
\text{bits}(128) & V_m = V[m]; \\
\text{bits}(128) & V_n = V[n]; \\
\text{bits}(128) & V_d = V[d]; \\
\text{bits}(32) & W_j\text{Prime}; \\
\text{bits}(32) & \text{result}; \\
\text{bits}(32) & SS2;
\end{align*}

\begin{align*}
W_j\text{Prime} &= \text{Elem}[V_m, i, 32]; \\
SS2 &= V_n<127:96> \text{ OR } \text{ROL}(V_d<127:96>, 12); \\
TT1 &= (V_d<127:96> \text{ AND } V_d<63:32>) \text{ OR } (V_d<127:96> \text{ AND } V_d<95:64>) \text{ OR } (V_d<63:32> \text{ AND } V_d<95:64>); \\
result<31:0> &= V_d<63:32>; \\
result<63:32> &= \text{ROL}(V_d<95:64>, 9); \\
result<95:64> &= V_d<127:96>; \\
V[d] &= \text{result};
\end{align*}
```

**Operational information**

If PSTATE.DIT is 1:
The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector,Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

### Advanced SIMD (FEAT_SM3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```c
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- `<imm2>` Is a 32-bit element indexed out of `<Vm>`, encoded in "imm2".

### Operation

```c
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bites(128) Vn = V[n];
bites(128) Vd = V[d];
bites(32) Wj;
bites(128) result;
bites(32) TT2;

Wj = Elem(Vm, i, 32);
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when **FEAT_SM3** is implemented.

**Advanced SIMD (FEAT_SM3)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rn</th>
<th>imm2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
</tr>
</tbody>
</table>

SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S<imm2>]

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

**Assembler Symbols**

- **<Vd>** is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- **<Vn>** is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- **<imm2>** is a 32-bit element indexed out of <Vm>, encoded in "imm2".

**Operation**

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = (Vd<127:96> AND Vd<95:64> OR (NOT(Vd<127:96>) AND Vd<63:32>));
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register. This instruction is implemented only when FEAT_SM4 is implemented.

Advanced SIMD (FEAT_SM4)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

SM4E <Vd>.4S, <Vn>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vn = V[n];
bits(32) intval;
bits(128) roundresult;
bits(32) roundkey;
roundresult = V[d];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SM4 is implemented.

Advanced SIMD
(FEAT_SM4)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vm = V[m];
bits(32) intval;
bits(128) result;
bits(32) const;
bits(128) roundresult;
roundresult = V[n];
for index = 0 to 3
   const = Elem[Vm, index, 32];
   intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
   for i = 0 to 3
      Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
   intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
   roundresult<31:0> = roundresult<63:32>;
   roundresult<63:32> = roundresult<95:64>;
   roundresult<95:64> = roundresult<127:96>;
   roundresult<127:96> = intval;
V[d] = roundresult;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem(operand1, e, esize), unsigned);
    element2 = Int(Elem(operand2, e, esize), unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

 integer d = UInt(Rd);
 integer n = UInt(Rn);
 integer m = UInt(Rm);
 if size == '11' then UNDEFINED;
 integer esize = 8 << UInt(size);
 integer datasize = if Q == '1' then 128 else 64;
 integer elements = datasize DIV esize;

 boolean unsigned = (U == '1');
 boolean minimum = (o1 == '1');

 Assembler Symbols

 <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
 <T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

 <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
 <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

 Operation

 bits(datasize) operand1 = V[n];
 bits(datasize) operand2 = V[m];
 bits(datasize) result;
 bits(2*datasize) concat = operand2:operand1;
 integer element1;
 integer element2;
 integer maxmin;

 for e = 0 to elements-1
   element1 = Int(Elem[concat, 2*e, esize], unsigned);
   element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
   maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
   Elem[result, e, esize] = maxmin<esize-1:0>;

 V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 1  | 1  | 1  | 0  | size| 1   | 1  | 0  | 0  | 0  | 0  | 0  | Rn | 1   | 0  | 1  | 0  | 1  | 0  |
| U   |    |    |    |    |    |    | op |     |    |    |    |    |    |    | Rd |     |    |    |    |    |    |

SMAXV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 * UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d] = maxmin<esize-1:0>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{center}
\begin{tabular}{cccccccccccccccc}
\hline
0 & O & 0 & 0 & 1 & 1 & 1 & 0 & \textbf{size} & 1 & \hline
Rm & 0 & 1 & 1 & 0 & 1 & 1 & \hline
Rn & \hline
Rd & 0 & \hline
\end{tabular}
\end{center}

\textbf{SMIN} $<$\textbf{Vd}$>$, $<$\textbf{Vn}$>$, $<$\textbf{Vm}$>$

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 \ll\hspace{1em} UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
\end{verbatim}

\section*{Assembler Symbols}

\begin{itemize}
    \item <\textbf{Vd}> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
    \item <\textbf{T}> Is an arrangement specifier, encoded in "size:Q":
    \begin{center}
    \begin{tabular}{cccc}
    \textbf{size} & \textbf{Q} & \textbf{<T>} \\
    \hline
    00 & 0 & 8B \\
    00 & 1 & 16B \\
    01 & 0 & 4H \\
    01 & 1 & 8H \\
    10 & 0 & 2S \\
    10 & 1 & 4S \\
    11 & x & RESERVED \\
    \end{tabular}
    \end{center}
    \item <\textbf{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
    \item <\textbf{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\end{itemize}

\section*{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 < UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

### Assembler Symbols

- **<V>** Is the destination width specifier, encoded in “size”:

  | size | <V>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- **<T>** Is an arrangement specifier, encoded in “size:Q”:

  | size | Q | <T>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
  element = Int(Elem[operand, e, esize], unsigned);
  maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d] = maxmin<esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register. The SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U | Q | 0 | 0 | 1 | 1 | 1 | size | L | M | Rm | 0 | 0 | 1 | 0 | H | 0 | Rn | Rd |

SMLAL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:
<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
ingteger element1;
ingteger element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register. The SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>Q</th>
<th>size</th>
<th>Rm</th>
<th>1 0</th>
<th>0 0</th>
<th>0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1</td>
<td>1 1</td>
<td>0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

SMLAL2 <Vd>.<Ta>, <Vn>.<Td>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  product = (element1*element2)<2*esize-1:0>;
  if sub_op then
    accum = Elem[operand3, e, 2*esize] - product;
  else
    accum = Elem[operand3, e, 2*esize] + product;
  Elem[result, e, 2*esize] = accum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register. The SMLSL2 instruction extracts vector elements from the upper half of the first source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | O  | O  | 0  | 1  | 1  | 1  | size| L  | M  | Rm |    | O  | 1  | 1  | 0  | H  | 0  | Rn |    | Rd |

**SMLSL{2} <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Ts>[<index>]**

```plaintext
text = integer idxdsize = if H == '1' then 128 else 64;
text = integer index;
text = bit Rmhi;
text = case size of
      text = when '01' index = UInt(H:L:M); Rmhi = '0';
text = when '10' index = UInt(H:L); Rmhi = M;
text = otherwise UNDEFINED;
text = integer d = UInt(Rd);
text = integer n = UInt(Rn);
text = integer m = UInt(Rmhi:Rm);
text = integer esize = 8 << UInt(size);
text = integer datasize = 64;
text = integer part = UInt(Q);
text = integer elements = datasize DIV esize;
text = boolean unsigned = (U == '1');
text = boolean sub_op = (o2 == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

SMLSL, SMLSL2 (by element)
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<size> <Vm> is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<index> <Ts> is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> <index> is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMLSL, SMLSL2 (vector)**

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register. The SMLSL2 instruction extracts each source vector from the upper half of each source register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>1</th>
<th>Rm</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**SMLSL{2} <Vd>, <Ta>, <Vn>, <Vm>**

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);

if size == '11' then UNDEFINED;
integer esize = \( 8 \ll \text{UInt}(size) \);
integer datasize = 64;
integer part = \( \text{UInt}(Q) \);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>**

Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>**

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Tb>**

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vm>**

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

**CheckFPAdvSIMDEnabled64();**

bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. \textit{ID\_AA64ISAR1\_EL1} I8MM indicates whether this instruction is supported.

Vector (FEAT\_I8MM)

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | \text{Rm} | 1 | 0 | 1 | 0 | 0 | 1 | \text{Rn} | \text{Rd} |
\end{verbatim}

SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

\begin{verbatim}
if !\text{HaveInt8MatMulExt}() then UNDEFINED;
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
integer d = \text{UInt}(Rd);
\end{verbatim}

Assembler Symbols

\begin{itemize}
\item \textless Vd\textgreater{} is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
\item \textless Vn\textgreater{} is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\item \textless Vm\textgreater{} is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\end{itemize}

Operation

\begin{verbatim}
\text{CheckFPAdvSIMDEnabled64}();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = \text{MatMulAdd}(addend, operand1, operand2, FALSE, FALSE);
\end{verbatim}

Internal version only: isa v33.16decrl, AdvSIMD v29.05, pseudocode v2021-12\_rel, sve v2021-12\_rel; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 32-bit (Q == 0)

\texttt{SMOV <Wd>, <Vn>.<Ts>[<index>]}

### 64-bit (Q == 1)

\texttt{SMOV <Xd>, <Vn>.<Ts>[<index>]}

integer \(d = \texttt{UInt}(Rd)\);
integer \(n = \texttt{UInt}(Rn)\);

inget\ integer\ size;\n\textbf{case} Q:imm5\ of\n\textbf{when} 'xxxxx1' size = 0;  // SMOV [WX]d, Vn.B  
\textbf{when} 'xxxx10' size = 1;  // SMOV [WX]d, Vn.H  
\textbf{when} '1xx100' size = 2;  // SMOV Xd, Vn.S  
\textbf{otherwise} UNDEFINED;

integer idxdsize = if \texttt{imm5<4>} == '1' then 128 else 64;
integer index = \texttt{UInt}(imm5<4:size+1>);
integer esize = 8 \ll\ size;
integer datasize = if Q == '1' then 64 else 32;

\textbf{Assembler Symbols}

\texttt{<Wd>} \quad \textbf{Is} the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\texttt{<Xd>} \quad \textbf{Is} the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\texttt{<Vn>} \quad \textbf{Is} the name of the SIMD&FP source register, encoded in the "Rn" field.

\texttt{<Ts>} \quad \textbf{For} the 32-bit variant: \textbf{is} an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>\texttt{imm5}</th>
<th>\texttt{&lt;Ts&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
</tbody>
</table>

\textbf{For} the 64-bit variant: \textbf{is} an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>\texttt{imm5}</th>
<th>\texttt{&lt;Ts&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
</tbody>
</table>

\texttt{<index>} \quad \textbf{For} the 32-bit variant: \textbf{is} the element index encoded in "imm5":

<table>
<thead>
<tr>
<th>\texttt{imm5}</th>
<th>\texttt{&lt;index&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>\texttt{imm5&lt;4:1&gt;}</td>
</tr>
<tr>
<td>xxxx10</td>
<td>\texttt{imm5&lt;4:2&gt;}</td>
</tr>
</tbody>
</table>
For the 64-bit variant: is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bv(idxsize) operand = V[n];
```

```c
X[d] = SignExtend(Elem[operand, index, esize], datasize);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register. The SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 0 | Q | 0 | 0 | 1 | 1 | 1 | size | L | M | Rm | 1 | 0 | 1 | 0 | H | 0 | Rn | Rd | U |
```

SMULL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:
<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

< Ts > Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt; Ts &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

< index > Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt; index &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMULL, SMULL2 (vector)**

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register. The SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 0  | size | 1  | Rm  | 1  | 1  | 0  | 0  | 0  | Rn  | Rd  |

**SMULL(2) \(<Vd>, <Ta>, <Vn>, <Vm>, <Tb>**

```plaintext
text
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "size":

```
size  <Ta>
00   8H
01   4S
10   2D
11   RESERVED
```

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

```
size  Q  <Tb>
00   0   8B
00   1   16B
01   0   4H
01   1   8H
10   0   2S
10   1   4S
11   x   RESERVED
```

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1 0 0 0 0 0 1 1 1 1 0</td>
<td>U</td>
</tr>
</tbody>
</table>

SQABS <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0</td>
<td>1 0 0 0 0 0 1 1 1 1 0</td>
<td>U</td>
</tr>
</tbody>
</table>

SQABS <Vd>..<T>, <Vn>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

\begin{verbatim}
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
\end{verbatim}

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQADD

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm  | 0  | 0  | 0  | 0  | 1  | 1  | Rn  | 0  | U  | Rd  |

SQADD <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0  | 0  | 1  | 1  | 1  | size | 1  | Rm  | 0  | 0  | 0  | 0  | 1  | 1  | Rn  | 0  | U  | Rd  |

SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLAL, SQDMLAL2 (by element)**

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLAL instruction extracts vector elements from the lower half of the first source register. The SQDMLAL2 instruction extracts vector elements from the upper half of the first source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 1 0 1 1 1 1 | size | L | M | Rm | 0 | 0 | 1 | 1 | H | 0 | Rn | Rd | 02
```

**SQDMLAL <Va><d>, <Vb><n>, <Vm><Ts>[<index>])**

```plaintext
integer idxds = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

### Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 0 0 1 1 1 | size | L | M | Rm | 0 | 0 | 1 | 1 | H | 0 | Rn | Rd | 02
```

SQDMLAL, SQDMLAL2 (by element)
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bite Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:  

SQDMLAL, SQDMLAL2 (by element)
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLAL, SQDMLAL2 (vector)**

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register. The SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 1</th>
<th>0 1 1 1 1 0</th>
<th>size</th>
<th>1</th>
<th>Rm</th>
<th>1 0 0 1 0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1 0 0 1 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQDMLAL <Va><d>, <Vb><n>, <Vb><m>**

```plaintext
text = UInt(Rd);
text = UInt(Rn);
text = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
text = 8 << UInt(size);
text = esize;
text = elements = 1;
text = part = 0;

boolean sub_op = (o1 == '1');
```

### Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | 0 1 | 1 1 1 0 | size | 1 | Rm | 1 0 0 1 0 0 | Rn | Rd |
|---|---|---|---|---|---|---|---|---|
| 0 Q 0 0 1 1 1 0 | size | 1 | Rm | 1 0 0 1 0 0 | Rn | Rd |
| 0 1 | o1 |

**SQDMLAL(2) <Vd><Ta>, <Vn><Tb>, <Vm><Tb>**

```plaintext
text = UInt(Rd);
text = UInt(Rn);
text = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
text = esize = 8 << UInt(size);
text = esize;
text = elements = esize;
text = part = 0;

boolean sub_op = (o1 == '1');
```

### Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

**SQDMLAL, SQDMLAL2**

*(vector)*
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element2 = SInt(Elem[operand2, e, esize]);
  (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
  if sub_op then
    accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
  else
    accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
  (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
  if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
**SQDMLSL, SQDMLSL2 (by element)**

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLSL instruction extracts vector elements from the lower half of the first source register. The SQDMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 size L M Rm 0 1 1 1 H 0 Rd
```

**SQDMLSL <Va><d>, <Vb><n>, <Vm..<Ts>[<index>])**

```plaintext
integer idxdsise = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

**Vector**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 size L M Rm 0 1 1 1 H 0 Rd
```

SQDMLSL, SQDMLSL2 (by element)
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

### Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”: 
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0::Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H::L::M</td>
</tr>
<tr>
<td>10</td>
<td>H::L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
binaryproduct;
binary accum;
boolean sat1;
boolean sat2;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLSL, SQDMLSL2 (vector)**

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The SQDMLSL instruction extracts each source vector from the lower half of each source register. The SQDMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQDMLSL** $<Va><d>, <Vb><n>, <Vb><m>**

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '00' || size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = esize;
- integer elements = 1;
- integer part = 0;
- boolean sub_op = (o1 == '1');

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQDMLSL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>**

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '00' || size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = 64;
- integer part = UInt(Q);
- integer elements = datasize DIV esize;
- boolean sub_op = (o1 == '1');

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”: 
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element2 = SInt(Elem[operand2, e, esize]);
  (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
  if sub_op then
    accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
  else
    accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
  (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
  if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULH (by element)

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.
The results are truncated. For rounded results, see SQRDMULH.
Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
size L M Rm 1 1 0 0 H 0 Rn Rd

SQDMULH <V<d>, <V<n>, <Vm><Ts>[<index>]>

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
size L M Rm 1 1 0 0 H 0 Rn Rd

SQDMULH <Vd><T>, <Vn><T>, <Vm>.<Ts>[<index>]>

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
**SQDMULH (vector)**

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see **SQRDMLUH**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

---

**Scalar**

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0 1 0 1 1 1 1 0 | size | 1 | Rm | 1 0 1 1 0 1 | Rn | Rd
```

```plaintext
SQDMULH <V><d>, <V><n>, <V><m>
```

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '11' || size == '00' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = esize;
- integer elements = 1;
- boolean rounding = (U == '1');

---

**Vector**

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0 1 0 1 1 1 1 0 | size | 1 | Rm | 1 0 1 1 0 1 | Rn | Rd
```

```plaintext
SQDMULH <Vd>..<T>, <Vn>..<T>, <Vm>..<T>
```

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '11' || size == '00' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean rounding = (U == '1');

---

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register. The SQDMULL2 instruction extracts the first source vector from the upper half of the first source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | L  | M  | Rm | 1  | 0  | 1  | 1  | H  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

SQDMULL <Va><d>, <Vb><n>, <Vm><Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | L  | M  | Rm | 1  | 0  | 1  | 1  | H  | 0  | Rn | Rd |

SQDMULL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

\[
\begin{array}{c|c}
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T_a>\) Is an arrangement specifier, encoded in “size”:

\[
\begin{array}{c|c}
00 & \text{RESERVED} \\
01 & 4S \\
10 & 2D \\
11 & \text{RESERVED} \\
\end{array}
\]

\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<T_b>\) Is an arrangement specifier, encoded in “size:Q”:

\[
\begin{array}{c|c|c}
00 & x & \text{RESERVED} \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED} \\
\end{array}
\]

\(<V_a>\) Is the destination width specifier, encoded in “size”:

\[
\begin{array}{c|c}
00 & \text{RESERVED} \\
01 & 5 \\
10 & 0 \\
11 & \text{RESERVED} \\
\end{array}
\]

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<V_b>\) Is the source width specifier, encoded in “size”:

\[
\begin{array}{c|c}
00 & \text{RESERVED} \\
01 & H \\
10 & S \\
11 & \text{RESERVED} \\
\end{array}
\]

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

\[
\begin{array}{c|c|c}
00 & \text{RESERVED} \\
01 & 0:M:Rm \\
10 & M:Rm \\
11 & \text{RESERVED} \\
\end{array}
\]

Restricted to V0-V15 when element size \(<Ts>\) is H.

\(<Ts>\) Is an element size specifier, encoded in “size”: 

SQDMULL, SQDMULL2 (by element)
<index> Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = $\text{SInt}(\text{Elem}[\text{operand2}, \text{index}, \text{esize}]);$
for e = 0 to elements-1
    element1 = $\text{SInt}(\text{Elem}[\text{operand1}, e, \text{esize}]);$
    (product, sat) = $\text{SignedSatQ}(2 \times \text{element1} \times \text{element2}, 2 \times \text{esize});$
    $\text{Elem}[\text{result}, e, 2\times\text{esize}] = \text{product};$
    if sat then FPSR.QC = '1';

$V[d] = \text{result};$
**SQDMULL, SQDMULL2 (vector)**

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes:Scalar and Vector

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>

SQDMULL <Va><d>, <Vb><n>, <Vb><m>

```plaintext
ingger d = UInt(Rd);
ingger n = UInt(Rn);
ingger m = UInt(Rm);
if size == '00' || size == '11' then UNDEFINED;
ingger esize = 8 << UInt(size);
ingger datasize = esize;
ingger elements = 1;
ingger part = 0;
```

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vn>.<Tb>

```plaintext
ingger d = UInt(Rd);
ingger n = UInt(Rn);
ingger m = UInt(Rm);
if size == '00' || size == '11' then UNDEFINED;
ingger esize = 8 << UInt(size);
ingger datasize = 64;
ingger part = UInt(Q);
ingger elements = datasize DIV esize;
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Q <Vd> Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<Ta> Is an arrangement specifier, encoded in “size”:
**size** | **<Ta>**
---|---
00 | RESERVED
01 | 4S
10 | 2D
11 | RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

| size | **Q** | **<Tb>** |
---|---|---|
00 | x | RESERVED
01 | 0 | 4H
01 | 1 | 8H
10 | 0 | 2S
10 | 1 | 4S
11 | x | RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<**Va**> Is the destination width specifier, encoded in "size":

| size | **<Va>** |
---|---|
00 | RESERVED
01 | S
10 | D
11 | RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vb**> Is the source width specifier, encoded in "size":

| size | **<Vb>** |
---|---|
00 | RESERVED
01 | H
10 | S
11 | RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bv256 operand1 = Vpart[n, part];
bv256 operand2 = Vpart[m, part];
bv256 result;
integer element1;
integer element2;
bv256 product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';
V[d] = result;
```

---

SQDMULL, SQDMULL2 (vector)

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQNEG**

Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 size 1 0 0 0 0 0 1 1 1 1 0 Rn Rd
```

**SQNEG <V><d>, <V><n>**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

**Vector**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 1 0 size 1 0 0 0 0 0 1 1 1 1 0 Rn Rd
```

**SQNEG <Vd>.<T>, <Vn>.<T>**

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

**Assembler Symbols**

**<V>** Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**<d>** Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<n>** Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{\textless} Vn > Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(ddatasize) operand = \text{\textgreater}V[n];
bits(ddatasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = \text{SInt}(\text{Elem}[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = \text{Abs}(element);
    (\text{Elem}[result, e, esize], sat) = \text{SignedSatQ}(element, esize);
    if sat then FPSR.QC = '1';
\text{\textless}V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

**Scalar**

(Feat_RDM)

```
|   | 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|--------------------------|--|----------|--|--------------------------|--|----------|
| S | 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 0 | Rn | Rd |
```

SQRDMLAH $<$V$><$d$>$, $<$V$><$n$>$, $<$Vm$>.<$Ts$>$[<index$>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

**Vector**

(Feat_RDM)

```
|   | 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|--------------------------|--|----------|--|--------------------------|--|----------|
| S | 0 0 1 0 1 1 1 1 1 1 0 1 0 1 1 0 | Rn | Rd |
```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”: 
<index> is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
  element1 = SInt(Elem[operand1, e, esize]);
  element3 = SInt(Elem[operand3, e, esize]);
  if sub_op then
    accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
  else
    accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
  (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
  if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar (FEAT_RDM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | size | 0  | Rm | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rn | 1  | 0  | 0  | 0  | 0  | 0  | Rd |
| S  |

**SQRDMLAH <V><d>, <V><n>, <V><m>**

```assembly
define SQRDMLAH if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' | size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

### Vector (FEAT_RDM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  | size | 0  | Rm | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rn | 1  | 0  | 0  | 0  | 0  | 0  | Rd |
| S  |

**SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

```assembly
define SQRDMLAH if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' | size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

### Assembler Symbols

**<V>** Is a width specifier, encoded in “size”:

---

SQRDMLAH (vector)
Is the number of the SIMD&FP destination register, in the "Rd" field.

<\text{n}> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<\text{Vd}> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\text{T}> Is an arrangement specifier, encoded in "size:Q":

\begin{tabular}{c|c|c}
\text{size} & <\text{V}> & <\text{T}> \\
--- & --- & --- \\
00 & RESERVED & \\
01 & H & \\
10 & S & \\
11 & RESERVED & \\
\end{tabular}

<\text{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

\texttt{CheckFPAdvSIMDEnabled64();}

\texttt{bits(datasize) operand1 = V[n];}
\texttt{bits(datasize) operand2 = V[m];}
\texttt{bits(datasize) operand3 = V[d];}
\texttt{bits(datasize) result;}
\texttt{integer rounding const = if rounding then 1 \ll (esize - 1) else 0;}
\texttt{integer element1;}
\texttt{integer element2;}
\texttt{integer element3;}
\texttt{integer product;}
\texttt{integer accum;}
\texttt{boolean sat;}
\texttt{for e = 0 to elements - 1}
\hspace{1em} \texttt{element1 = SInt(Elem[operand1, e, esize]);}
\hspace{1em} \texttt{element2 = SInt(Elem[operand2, e, esize]);}
\hspace{1em} \texttt{element3 = SInt(Elem[operand3, e, esize]);}
\hspace{1em} \texttt{if sub_op then}
\hspace{2em} \texttt{accum = ((element3 \ll esize) - 2 \ast (element1 \ast element2) + rounding const);}
\hspace{1em} \texttt{else}
\hspace{2em} \texttt{accum = ((element3 \ll esize) + 2 \ast (element1 \ast element2) + rounding const);}
\hspace{1em} \texttt{(Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);}
\hspace{1em} \texttt{if sat then FPSR.QC = '1';}
\texttt{V[d] = result;}

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |

SQRDMLSH \(<V><d>\), \(<V><n>, \(<V\.T_s>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = Uint(H:L:M); Rmhi = '0';
  when '10' index = Uint(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rmhi:Rm);

integer esize = 8 << Uint(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector

(FEAT_RDM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:
Size

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_RDM)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 | size | 0 | Rd | 1 0 0 0 0 1 1 | Rn | Rm | S

SQRDMLSH <V><d>, <V><n>, <V><m>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(FEAT_RDM)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 0 | size | 0 | Rd | 1 0 0 0 0 1 1 | Rn | Rm | S

SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m =UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size.Q":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQRDMULH.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 1</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
Assembler Symbols

<\textless{}V\textgreater{>} Is a width specifier, encoded in "size":

\begin{center}
\begin{tabular}{c|c}
\textbf{size} & \textbf{<\textless{}V\textgreater{>}} \\
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\end{tabular}
\end{center}

<\textless{}d\textgreater{} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<\textless{}n\textgreater{} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<\textless{}Vd\textgreater{} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\textless{}T\textgreater{} Is an arrangement specifier, encoded in "size:Q":

\begin{center}
\begin{tabular}{c|c|c}
\textbf{size} & \textbf{Q} & \textbf{<\textless{}T\textgreater{>}} \\
\hline
00 & x & RESERVED \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\end{tabular}
\end{center}

<\textless{}Vn\textgreater{} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\textless{}Vm\textgreater{} Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

\begin{center}
\begin{tabular}{c|c}
\textbf{size} & \textbf{<\textless{}Vm\textgreater{>}} \\
\hline
00 & RESERVED \\
01 & 0:Rm \\
10 & M:Rm \\
11 & RESERVED \\
\end{tabular}
\end{center}

Restricted to V0-V15 when element size <\textless{}Ts\textgreater{} is H.

<\textless{}Ts\textgreater{} Is an element size specifier, encoded in "size":

\begin{center}
\begin{tabular}{c|c}
\textbf{size} & \textbf{<\textless{}Ts\textgreater{>}} \\
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\end{tabular}
\end{center}

<\textless{}index\textgreater{} Is the element index, encoded in "size:L:H:M":

\begin{center}
\begin{tabular}{c|c}
\textbf{size} & \textbf{<\textless{}index\textgreater{>}} \\
\hline
00 & RESERVED \\
01 & H:L:M \\
10 & H:L \\
11 & RESERVED \\
\end{tabular}
\end{center}
Operation

\[
\text{CheckFPAdvSIMDEnabled64}(\); \\
\text{bits}(\text{datasize}) \text{ operand1} = V[n]; \\
\text{bits}(\text{idxsize}) \text{ operand2} = V[m]; \\
\text{bits}(\text{datasize}) \text{ result}; \\
\text{integer} \text{ round\_const} = \text{if round then } 1 << (\text{esize} - 1) \text{ else } 0; \\
\text{integer} \text{ element1}; \\
\text{integer} \text{ element2}; \\
\text{integer} \text{ product}; \\
\text{boolean} \text{ sat}; \\
\]

\[
\text{element2} = \text{Sint}(\text{Elem}[\text{operand2}, \text{index}, \text{esize}]); \\
\text{for e = 0 to elements-1} \\
\begin{align*}
\text{element1} &= \text{Sint}(\text{Elem}[\text{operand1}, \text{e}, \text{esize}]); \\
\text{product} &= (2 * \text{element1} * \text{element2}) + \text{round\_const}; \\
\text{if} \text{ sat} \text{ then} \text{FPSR.QC} &= \text{'}1\text{'};
\end{align*}
\]

\[
\text{V[d]} = \text{result};
\]
SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see `SQDMULH`.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit `FPSR.QC` is set.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 1 1 1 1 1 0 size 1 Rm 1 0 1 0 1 Rd
```

\[
\text{SQRDMULH} \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle, \langle V \rangle \langle m \rangle
\]

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');

### Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 0 1 1 1 0 size 1 Rm 1 0 1 0 1 Rd
```

\[
\text{SQRDMULH} \langle Vd \rangle \langle T \rangle, \langle Vn \rangle \langle T \rangle, \langleVm \rangle \langle T \rangle
\]

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');

### Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<m>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

Scalar

<table>
<thead>
<tr>
<th>d</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

<table>
<thead>
<tr>
<th>d</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
</tr>
</tbody>
</table>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQRSHRN.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | != | 0000 | immb | 1  | 0  | 0  | 1  | 1  | Rn | Rd |

U   immh   op

SQRSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | != | 0000 | immb | 1  | 0  | 0  | 1  | 1  | Rn | Rd |

U   immh   op

SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>08H</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>04H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>08H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in \( \text{"immh:immb"} \):

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>SFF Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQRSHRUN.

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

### Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 1 1 0 | != 0000 | immh | 1 0 0 1 | Rd |

SQRSHRUN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

### Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 0 1 1 1 0 | != 0000 | immh | 1 0 0 1 | Rd |

SQRSHRUN2 {2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

### Assembler Symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q".
<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>  Is an arrangement specifier, encoded in “immm:Q”:

<table>
<thead>
<tr>
<th>immm</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>  Is an arrangement specifier, encoded in “immm”:

<table>
<thead>
<tr>
<th>immm</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>  Is the destination width specifier, encoded in “immm”:

<table>
<thead>
<tr>
<th>immm</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>   Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>  Is the source width specifier, encoded in “immm”:

<table>
<thead>
<tr>
<th>immm</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>   Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immm:immb”:

<table>
<thead>
<tr>
<th>immm</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immm:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immm:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immm:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immm:immb”: 
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSHL (immediate)

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 0 1 0 1 1 1 1 1 0 != 0000 immb 0 1 1 1 0 1</td>
</tr>
</tbody>
</table>

SQSHL <V><d>, <V><n>, #<shift>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 0 0 0 1 1 1 1 1 0 != 0000 immh 0 1 1 1 0 1</td>
</tr>
</tbody>
</table>

SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
  element = Int(Elem[operand, e, esize], src_unsigned) << shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
  if sat then FPSR.QC = '1';
V[d] = result;
```
SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>0 1 0</th>
<th>0 1 1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>0 1 0</th>
<th>0 1 1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHLU**

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see **UQRSHL**.

If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**.

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 1 1</th>
<th>1 1 1 1 1 1 0</th>
<th>!= 0000</th>
<th>immh</th>
<th>0 1 1</th>
<th>0 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQSHLU <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 1</th>
<th>0 1 1 1 1 1 0</th>
<th>!= 0000</th>
<th>immh</th>
<th>0 1 1</th>
<th>0 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
Assembler Symbols

<V> Is a width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
**SQSHRN, SQSHRN2**

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see *SQRSRN*.

The **SQSHRN** instruction writes the vector to the lower half of the destination register and clears the upper half, while the **SQSHRN2** instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR.QC* is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 | 0 1 1 1 1 1 0 | != 0000 | immh | 1 0 0 1 | 0 1 | Rn | Rd | op
```

```
SQSHRN <Vb><d>, <Va><n>, #<shift>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

**Vector**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 0 1 1 1 1 1 0 | != 0000 | immh | 1 0 0 1 | 0 1 | Rn | Rd | op
```

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
Assembler Symbols

2. Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8H</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>011x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>011x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SFF_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHRUN, SQSHRUN2**

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSRUN.

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|
| immh              | Rd                | Rn                |
| 0 1 1 1 1 1 1 1 1 0 | ! = 0000          | immb              | 1 0 0 0 0 1 |
```

**SQSHRUN** `<Vb><d>, <Va><n>, #<shift>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

### Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|
| immh              | Rd                | Rn                |
| 0 Q 1 0 1 1 1 1 1 0 | ! = 0000          | immb              | 1 0 0 0 0 1 |
```

**SQSHRUN[2]** `<Vd>.<Tb>, <Vn>.<Ta>, #<shift>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

---

SQSHRUN, SQSHRUN2 Page 1465
The name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the destination width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the source width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
### immh <shift>

<table>
<thead>
<tr>
<th>immh</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

---

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSUB**

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 1 1 1 1 0 | size | 1 | Rm | 0 0 1 0 1 1 | Rn | Rd |
  U
```

**SQSUB** <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

**Vector**

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 1 1 1 1 0 | size | 1 | Rm | 0 0 1 0 1 1 | Rn | Rd |
  U
```

**SQSUB** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQXTN, SQXTN2**

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

---

**Scalar**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 0 1 1 1 1 0 | size | 1 0 0 0 0 | 1 0 1 0 0 | 1 0 | Rn | Rd |
| U |
```

**SQXTN <Vb><d>, <Va><n>**

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
boolean unsigned = (U == '1');
```

---

**Vector**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 | Q | 0 1 1 1 1 0 | size | 1 0 0 0 0 | 1 0 1 0 0 | 1 0 | Rn | Rd |
| U |
```

**SQXTN2 <Vd>.<Tb>, <Vn>.<Ta>**

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

---

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:
<Q>
| 0 | [absent] |
| 1 | [present] |

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>  Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>  Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>  Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va>  Is the source width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
b不失2*datasize operand = V[n];
b不失(datasize) result;
b不失(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int)(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQXTUN, SQXTUN2**

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSP.QC is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0 1 0 0 1 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQXTUN** `<Vb><d>, <Va><n>`

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0 1 0 0 1 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**SQXTUN[2]** `<Vd>.<Tb>, <Vn>.<Ta>`

```java
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler Symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>  Is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bills(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SRHADD**

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see **SHADD**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | Q   | 0   | 0   | 1   | 1   | 0   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

**SRHADD** `<Vd>..<T>`, `<Vn>..<T>`, `<Vm>..<T>`

```cpp
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.

Depending on the settings in the \texttt{CPACR_EL1}, \texttt{CPTR_EL2}, and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}

\textbf{Scalar}

\begin{verbatim}
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

\hline
 immh & ! = 0000 & immh & 0 & 1 & 0 & 0 & 0 & \text{Rn} & \text{Rd} \\
\hline
\end{verbatim}

\texttt{SRI <V>d>, <V>n>, \#<shift>}

\begin{verbatim}
  integer d = UInt(Rd);
  integer n = UInt(Rn);
  if immh<3> != '1' then UNDEFINED;
  integer esize = 8 << 3;
  integer datasize = esize;
  integer elements = 1;
  integer shift = (esize * 2) - UInt(immh:immb);
\end{verbatim}

\textbf{Vector}

\begin{verbatim}
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

\hline
 immh & ! = 0000 & immh & 0 & 1 & 0 & 0 & 0 & \text{Rn} & \text{Rd} \\
\hline
\end{verbatim}

\texttt{SRI <Vd>.<T>, <Vn>.<T>, \#<shift>}

\begin{verbatim}
  integer d = UInt(Rd);
  integer n = UInt(Rn);
  if immh == '0000' then \texttt{SEE(asmidimm)};
  if immh<3>:Q == '10' then UNDEFINED;
  integer esize = 8 << \texttt{HighestSetBit}(immh);
  integer datasize = if Q == '1' then 128 else 64;
  integer elements = datasize DIV esize;
  integer shift = (esize * 2) - UInt(immh:immb);
\end{verbatim}
Assembler Symbols

<\text{V}> \quad \text{Is a width specifier, encoded in “immh”:}

<table>
<thead>
<tr>
<th>immh</th>
<th>\text{&lt;V&gt; }</th>
<th>0\text{xx}</th>
<th>\text{RESERVED}</th>
<th>1\text{xx}</th>
<th>D</th>
</tr>
</thead>
</table>

<d> \quad \text{Is the number of the SIMD&FP destination register, in the “Rd” field.}

<n> \quad \text{Is the number of the first SIMD&FP source register, encoded in the "Rn" field.}

<Vd> \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}

<T> \quad \text{Is an arrangement specifier, encoded in “immh:Q”:}

<table>
<thead>
<tr>
<th>immh</th>
<th>\text{Q}</th>
<th>\text{&lt;T&gt; }</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>\text{SEE Advanced SIMD modified immediate}</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>0H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>\text{RESERVED}</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}

<\text{shift}> \quad \text{For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:}

<table>
<thead>
<tr>
<th>immh</th>
<th>\text{&lt;shift&gt; }</th>
<th>0\text{xx}</th>
<th>\text{RESERVED}</th>
<th>1\text{xx}</th>
<th>(128-\text{UInt}(immh:immb))</th>
</tr>
</thead>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>\text{&lt;shift&gt; }</th>
<th>0000</th>
<th>\text{SEE Advanced SIMD modified immediate}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(immh:immb))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(immh:immb))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(immh:immb))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lxxx</td>
<td>(128-\text{UInt}(immh:immb))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation

\text{CheckFPAdvSIMDEnabled64}();
\text{bits(datasize) operand} = \text{\texttt{V}[n];}
\text{bits(datasize) operand2} = \text{\texttt{V}[d];}
\text{bits(datasize) result;}
\text{bits(esize) mask} = \text{\texttt{LSR(Ones(esize), shift)};}
\text{bits(esize) shifted;}
\text{for e = 0 to elements-1}
\text{ shifted} = \text{\texttt{LSR(Elem[operand, e, esize], shift)};}
\text{ Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;}
\text{V[d] = result;}

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see \texttt{SSHL}.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}

### Scalar

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm   | 0 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |

\texttt{SRSHL} <\texttt{V}> \texttt{<d>}, <\texttt{V}> \texttt{<n>}, <\texttt{V}> \texttt{<m>}
```

```plaintext
integer\ d = UInt(Rd);
integer\ n = UInt(Rn);
integer\ m = UInt(Rm);
integer\ esize = 8 \* UInt(size);
integer\ datasize = esize;
integer\ elements = 1;
boolean\ unsigned = (U == '1');
boolean\ rounding = (R == '1');
boolean\ saturating = (S == '1');
if S == '0' \&\& size != '11' then UNDEFINED;
```

### Vector

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
| 0 | Q | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm   | 0 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |

\texttt{SRSHL} <\texttt{Vd}> \texttt{<T>}, <\texttt{Vn}> \texttt{<T>}, <\texttt{Vm}> \texttt{<T>}
```

```plaintext
integer\ d = UInt(Rd);
integer\ n = UInt(Rn);
integer\ m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer\ esize = 8 \* UInt(size);
integer\ datasize = if Q == '1' then 128 else 64;
integer\ elements = datasize DIV esize;
boolean\ unsigned = (U == '1');
boolean\ rounding = (R == '1');
boolean\ saturating = (S == '1');
```

### Assembler Symbols

<\texttt{V}> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\texttt{V}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<\texttt{d}> Is the number of the SIMD&FP destination register, in the “Rd” field.

<\texttt{n}> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see **SSHR**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

\[
\begin{array}{cccccccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 0 & \text{!= 0000} & \text{immb} & 0 & 0 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rd} \\
\text{U} & \text{immh} & 01 & 00
\end{array}
\]

SRSHR **<V>d>**, **<V><n>**, $#<\text{shift}>$

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - \text{UInt}(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Vector**

\[
\begin{array}{cccccccccccccccccc}
0 & Q & 0 & 1 & 1 & 1 & 0 & \text{!= 0000} & \text{immb} & 0 & 0 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rd} \\
\text{U} & \text{immh} & 01 & 00
\end{array}
\]

SRSHR **<Vd>.<T>**, **<Vn>.<T>**, $#<\text{shift}>$

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

if immh == '0000' then \text{SEE(asimdimm)};
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << \text{HighestSetBit}(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - \text{UInt}(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler Symbols**

\[<V>\] Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

SRSHR
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|
| 0 1 1 1 1 1 1 0 | != 0000          | immh | 0 0 1 1 0 1 |
| U                | immh             | Rd    |
```

**SRSRA** `<V><d>, <V><n>, #<shift>`

integer `d = UInt(Rd)`;
integer `n = UInt(Rn)`;

if `immh<3> != '1'` then UNDEFINED;
integer `esize = 8 << 3`;
integer `datasize = esize`;
integer `elements = 1`;

integer `shift = (esize * 2) - UInt(immh:immb)`;
boolean `unsigned = (U == '1')`;
boolean `round = (o1 == '1')`;
boolean `accumulate = (o0 == '1')`;

### Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|
| 0 | O | 0 1 1 1 1 1 0 | != 0000          | immh | 0 0 1 1 0 1 |
| U                | immh             | Rd    |
```

**SRSRA** `<Vd>..<T>, <Vn>..<T>, #<shift>`

integer `d = UInt(Rd)`;
integer `n = UInt(Rn)`;

if `immh == '0000'` then SEE(asimdimm);
if `immh<3>:Q == '10'` then UNDEFINED;
integer `esize = 8 << HighestSetBit(immh)`;
integer `datasize = if Q == '1' then 128 else 64`;
integer `elements = datasize DIV esize`;

integer `shift = (esize * 2) - UInt(immh:immb)`;
boolean `unsigned = (U == '1')`;
boolean `round = (o1 == '1')`;
boolean `accumulate = (o0 == '1')`;

### Assembler Symbols

| `<V>`         | Is a width specifier, encoded in “immh”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>immh</strong></td>
<td><code>&lt;V&gt;</code></td>
</tr>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>lxxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>lxxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>lxxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>lxxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bvbit(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see SRSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>0 1 0 1 1 1 1 0</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

SSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>0 1 0 1 1 1 1 0</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 0 1</td>
<td>Rn</td>
</tr>
</tbody>
</table>

SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SSHLL, SSHLL2**

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register. The SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | Q  | 0 | 1 | 1 | 1 | 0 | != 0000 | immmb | 1 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |

**SSHLL(2) <Vd>.<Ta>, <Vn>.<Tb>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>lxxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”: 
<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTL, SXTL2</td>
<td>immh == '000' &amp;&amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

SSHR <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

SSHR <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>011x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>011x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>011x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 1 0 1 0 0 0 1 0 1 Rn Rd

SSRA <V>d, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 0 1 Rn Rd

SSRA <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>D</td>
</tr>
</tbody>
</table>

Page 1490
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>011x</td>
<td>0 4H</td>
</tr>
<tr>
<td>0001</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>011x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
b bits(datasize) operand2;
b bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

---

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register. The SSUBL2 instruction extracts each source vector from the upper half of each source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

SSUBL2(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
\[d\] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register. The SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
<th>Rm</th>
<th>size</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

SSUBW2 <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | x | size | Rn | | Rt | L | opcode |

One register (opcode == 0111)

ST1 { <Vt>.<T> }, [Xn|SP]

Two registers (opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [Xn|SP]

Three registers (opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP]

Four registers (opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [Xn|SP]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | Rm | x | x | 1 | x | size | Rn | | Rt | L | opcode |
One register, immediate offset (Rm == 11111 && opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>

One register, register offset (Rm != 11111 && opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Two registers, register offset (Rm != 11111 && opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Three registers, register offset (Rm != 11111 && opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Four registers, register offset (Rm != 11111 && opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

### Shared Decode

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;   // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;   // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;   // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;   // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;   // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;   // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;   // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;   // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST1 (multiple structures)  Page 1498
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a
SIMD&FP register to memory.
Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |

8-bit (opcode == 000)

ST1 { <Vt>.B }[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

ST1 { <Vt>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

ST1 { <Vt>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

ST1 { <Vt>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |
8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1 { <Vt>.B}[<index>], [<Xn|SP>], #1
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1 { <Vt>.B}[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H}[<index>], [<Xn|SP>], #2
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H}[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S}[<index>], [<Xn|SP>], #4
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S}[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D}[<index>], [<Xn|SP>], #8
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D}[<index>], [<Xn|SP>], <Xm>
```

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

### Assembler Symbols

- `<Vt>` is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<index>` is:
  - For the 8-bit variant: the element index, encoded in "Q:S:size".
  - For the 16-bit variant: the element index, encoded in "Q:S:size<1>".
  - For the 32-bit variant: the element index, encoded in "Q:S".
  - For the 64-bit variant: the element index, encoded in "Q".
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
case scale of
  when 3  // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
      scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;

ST1 (single structure)
Operation

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the \texttt{CPACR\_EL1, CPTR\_EL2,} and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{No offset} and \texttt{Post-index}

\textbf{No offset}

\begin{verbatim}
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rn | Rt |
\hline
0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rn | Rt |
\end{verbatim}

\texttt{ST2 \{ <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>]

\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
\end{verbatim}

\textbf{Post-index}

\begin{verbatim}
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rm | Rn | Rt |
\hline
0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rn | Rt |
\end{verbatim}

\texttt{Immediate offset (Rm == 11111)}

\texttt{ST2 \{ <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>], <imm>

\texttt{Register offset (Rm != 11111)}

\texttt{ST2 \{ <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>], <Xm>

\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
\end{verbatim}

\textbf{Assembler Symbols}

\texttt{<Vt>} Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

\texttt{<T>} Is an arrangement specifier, encoded in "size:Q":

\begin{verbatim}
| size | Q | <T> |
\hline
00 | 0 | 8B |
00 | 1 | 16B |
01 | 0 | 4H |
01 | 1 | 8H |
10 | 0 | 2S |
10 | 1 | 4S |
11 | 0 | RESERVED |
11 | 1 | 2D |
\end{verbatim}

\texttt{<Vt2>} Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

\texttt{<Xn|SP>} Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;   // number of iterations
integer selem; // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;   // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;   // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;   // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;   // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;   // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;   // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;   // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST2 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>S</td>
<td>size</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L R       opcode

8-bit (opcode == 000)

ST2 {<Vt>.B, <Vt2>.B}[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

ST2 {<Vt>.H, <Vt2>.H}[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

ST2 {<Vt>.S, <Vt2>.S}[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

ST2 {<Vt>.D, <Vt2>.D}[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>S</td>
<td>size</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L R       opcode
8-bit, immediate offset (Rm == 11111 && opcode == 000)

ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)

ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = Uint(Rt);
integer n = Uint(Rn);
integer m = Uint(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);  // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);  // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);  // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);  // D[0-1]
            scale = 3;
    endcase

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
  CheckSPAlignment();
  address = SP[ ];
else
  address = X[n];
offs = Zeros();
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
  // load/store one element per register
  for s = 0 to selem-1
    rval = V[t];
    if memop == MemOp_LOAD then
      // insert into one lane of 128-bit register
      Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
      V[t] = rval;
    else // memop == MemOp_STORE
      // extract from one lane of 128-bit register
      Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
      offs = offs + ebytes;
      t = (t + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[ ] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | size | Rn | Rt |

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | size | Rm | 0 | 1 | 0 | 0 | size | Rn | Rt |

Immediate offset (Rm == 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <imm>

Register offset (Rm != 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [Xn|SP], <Xm>

Assembler Symbols

<vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST3 (multiple structures)
Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;
            end if
        end for
    end for
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
end if
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn |   |   | Rt |
| L  | R  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

8-bit (opcode == 001)

```
```

16-bit (opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn |   |   | Rt |
| L  | R  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```
8-bit, immediate offset (Rm == 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if `HaveMTE2Ext()` then
    `SetTagCheckedInstruction(tag_checked);`

`CheckFPAdvSIMDEnabled64();`

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    `CheckSPAlignment();`
    address = `SP[]`;
else
    address = `X[n];`

offs = `Zeros();`
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = `Mem[address+offs, ebytes, AccType_VEC];`
    // replicate to fill 128- or 64-bit register
    `V[t] = Replicate(element, datasize DIV esize);`
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = `V[t];`
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            `Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];`
            `V[t] = rval;`
        else
            // memop == MemOp_STORE
            `Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];`
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = `X[m];`
    if n == 31 then
        `SP[] = address + offs;`
    else
        `X[n] = address + offs;`

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: No offset and Post-index.

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | size | Rn | Rt |

L  opcode


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | size | Rm | Rn | Rt |

L  opcode

Immediate offset (Rm == 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```pasm
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem; // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST4 (multiple structures)
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. It has encodings from 2 classes: No offset and Post-index

No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 1 1 0 1 0 0 0 0 0 0 x x 1 | S | size | Rn | Rt |
| L | R | opcode |

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 1 1 0 1 1 0 1 0 0 0 x x 1 | S | size | Rn | Rt |
| L | R | opcode |
8-bit, immediate offset (Rm == 11111 && opcode == 001)


8-bit, register offset (Rm != 11111 && opcode == 001)


16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)


16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)


32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)


32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)


64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)


64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt>    Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>   Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>   Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>   Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in “Q:S:size”.
For the 16-bit variant: is the element index, encoded in “Q:S:size<1>”.
For the 32-bit variant: is the element index, encoded in “Q:S”.
For the 64-bit variant: is the element index, encoded in “Q”.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>    Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);  
integer scale = init_scale;  
integer selem = UInt(opcode<0>:R) + 1;  
boolean replicate = FALSE;  
integer index;  

case scale of  
  when 3  
    // load and replicate  
    if L == '0' || S == '1' then UNDEFINED;  
    scale = UInt(size);  
    replicate = TRUE;  
  when 0  
    index = UInt(Q:S:size);  // B[0-15]  
  when 1  
    if size<0> == '1' then UNDEFINED;  
    index = UInt(Q:S:size<1>);  // H[0-7]  
  when 2  
    if size<1> == '1' then UNDEFINED;  
    if size<0> == '0' then  
      index = UInt(Q:S);  // S[0-3]  
    else  
      if S == '1' then UNDEFINED;  
      index = UInt(Q);  // D[0-1]  
    scale = 3;  

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;  
integer datasize = if Q == '1' then 128 else 64;  
integer esize = 8 << scale;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point Non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit (opc == 00)

STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]  

64-bit (opc == 01)

STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]  

128-bit (opc == 10)

STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]  

// Empty.

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
Operation

```c
CheckFPAdvSIMEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
else
    if n == 31 then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm7</td>
<td></td>
<td>Rt2</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm7</td>
<td></td>
<td>Rt2</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>]

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>]

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>]

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm7</td>
<td></td>
<td>Rt2</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit (opc == 00)

\[ \text{STP} \ <St1>, \ <St2>, \ [<Xn|SP>\{, \#<imm>\}] \]

64-bit (opc == 01)

\[ \text{STP} \ <Dt1>, \ <Dt2>, \ [<Xn|SP>\{, \#<imm>\}] \]

128-bit (opc == 10)

\[ \text{STP} \ <Qt1>, \ <Qt2>, \ [<Xn|SP>\{, \#<imm>\}] \]

boolean \ wback = \text{FALSE};
boolean \ postindex = \text{FALSE};

**Assembler Symbols**

\(<Dt1>\) \quad \text{Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.}
\(<Dt2>\) \quad \text{Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.}
\(<Qt1>\) \quad \text{Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.}
\(<Qt2>\) \quad \text{Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.}
\(<St1>\) \quad \text{Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.}
\(<St2>\) \quad \text{Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.}
\(<Xn|SP>\) \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}
\(<imm>\) \quad \text{For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.}
\quad \text{For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.}
\quad \text{For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.}
\quad \text{For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.}
\quad \text{For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.}
\quad \text{For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.}

**Shared Decode**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bias(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
```
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VEC] = data1;
Mem[address+dbytes, dbytes, AccType_VEC] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

```
<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>imm9</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

8-bit (size == 00 & opc == 00)

```
STR <Bt>, [<Xn|SP>], #<simm>
```

16-bit (size == 01 & opc == 00)

```
STR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 & opc == 00)

```
STR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 & opc == 00)

```
STR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 & opc == 10)

```
STR <Qt>, [<Xn|SP>], #<simm>
```

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

```
<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>imm9</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
8-bit (size == 00 && opc == 00)
STR <Bt>, [<Xn|SP>, #<simm>]

16-bit (size == 01 && opc == 00)
STR <Ht>, [<Xn|SP>, #<simm>]

32-bit (size == 10 && opc == 00)
STR <St>, [<Xn|SP>, #<simm>]

64-bit (size == 11 && opc == 00)
STR <Dt>, [<Xn|SP>, #<simm>]

128-bit (size == 00 && opc == 10)
STR <Qt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    | x  |    |    |    |    |    | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

8-bit (size == 00 && opc == 00)
STR <Bt>, [<Xn|SP>{, #<pimm}>]

16-bit (size == 01 && opc == 00)
STR <Ht>, [<Xn|SP>{, #<pimm}>]

32-bit (size == 10 && opc == 00)
STR <St>, [<Xn|SP>{, #<pimm}>]

64-bit (size == 11 && opc == 00)
STR <Dt>, [<Xn|SP>{, #<pimm}>]

128-bit (size == 00 && opc == 10)
STR <Qt>, [<Xn|SP>{, #<pimm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
Assembler Symbols

<Bt>  Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt>  Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht>  Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt>  Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St>  Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```
Operation

CheckFPAdvSIMDEnabled64();
bias(64) address;
bias(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rm</td>
<td>option</td>
<td>S</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 00 && option != 011)

\[ \text{STR} \langle Bt \rangle, \lbrack \langle Xn|SP \rangle, \langle Wm \rangle | \langle Xm \rangle \rbrack, \langle \text{extend} \rangle \{\langle \text{amount} \rangle \} \]

8-bit (size == 00 && opc == 00 && option == 011)

\[ \text{STR} \langle Bt \rangle, \lbrack \langle Xn|SP \rangle, \langle Xm \rangle \{, \text{LSL} \langle \text{amount} \rangle \} \]

16-bit (size == 01 && opc == 00)

\[ \text{STR} \langle Ht \rangle, \lbrack \langle Xn|SP \rangle, \langle Wm \rangle | \langle Xm \rangle \rbrack\{, \langle \text{extend} \rangle \{\langle \text{amount} \rangle \} \}

32-bit (size == 10 && opc == 00)

\[ \text{STR} \langle St \rangle, \lbrack \langle Xn|SP \rangle, \langle Wm \rangle | \langle Xm \rangle \rbrack\{, \langle \text{extend} \rangle \{\langle \text{amount} \rangle \} \}

64-bit (size == 11 && opc == 00)

\[ \text{STR} \langle Dt \rangle, \lbrack \langle Xn|SP \rangle, \langle Wm \rangle | \langle Xm \rangle \rbrack\{, \langle \text{extend} \rangle \{\langle \text{amount} \rangle \} \}

128-bit (size == 00 && opc == 10)

\[ \text{STR} \langle Qt \rangle, \lbrack \langle Xn|SP \rangle, \langle Wm \rangle | \langle Xm \rangle \rbrack\{, \langle \text{extend} \rangle \{\langle \text{amount} \rangle \} \}

integer scale = \text{UInt}(\text{opc} < 1>:\text{size});
if scale > 4 then UNDEFINED;
if option < 1 > == '0' then UNDEFINED; // sub-word index
\text{ExtendType} \text{ extend_type} = \text{DecodeRegExtend}(\text{option});
integer shift = if S == '1' then scale else 0;

Assembler Symbols

\<Bt> \quad \text{Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.}

\<Dt> \quad \text{Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.}

\<Ht> \quad \text{Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.}

\<Qt> \quad \text{Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.}

\<St> \quad \text{Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.}

\langle Xn|SP \rangle \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}

\langle Wm \rangle \quad \text{When option < 0 > is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.}

\langle Xm \rangle \quad \text{When option < 0 > is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.}
For the 8-bit variant: is the index extend specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#4</td>
</tr>
</tbody>
</table>

**Shared Decode**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

bits(64) offset = \texttt{ExtendReg}(m, extend\_type, shift);
\texttt{CheckFPAdvSIMDEnabled64}();
bits(64) address;
bits(datasize) data;

if \texttt{HaveMTE2Ext}() then
   \texttt{SetTagCheckedInstruction}(tag\_checked);

if n == 31 then
   \texttt{CheckSPAlignment}();
   address = \texttt{SP}[];
else
   address = \texttt{X}[n];

address = address + offset;

\texttt{case memop of}
   \texttt{when MemOp\_STORE}
      data = \texttt{V}[t];
      \texttt{Mem}[address, datasize\ \text{DIV} 8, AccType\_VEC] = data;
   \texttt{when MemOp\_LOAD}
      data = \texttt{Mem}[address, datasize\ \text{DIV} 8, AccType\_VEC];
      \texttt{V}[t] = data;

Operational information

If \texttt{PSTATE.DIT} is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

8-bit (size == 00 && opc == 00)

STUR \(<Bt>, \{<Xn|SP\\}, \#<simm>\}"

16-bit (size == 01 && opc == 00)

STUR \(<Ht>, \{<Xn|SP\\}, \#<simm>\}"

32-bit (size == 10 && opc == 00)

STUR \(<St>, \{<Xn|SP\\}, \#<simm>\}"

64-bit (size == 11 && opc == 00)

STUR \(<Dt>, \{<Xn|SP\\}, \#<simm>\}"

128-bit (size == 00 && opc == 10)

STUR \(<Qt>, \{<Xn|SP\\}, \#<simm>\}"

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

\(<Bt>\) Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Dt>\) Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Ht>\) Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Qt>\) Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<St>\) Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<simm>\) Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
\texttt{MemOp} memop = if opc<0> == '1' then \texttt{MemOp\_LOAD} else \texttt{MemOp\_STORE};
integer datasize = 8 << scale;
boolean tag_checked = memop != \texttt{MemOp\_PREFETCH} && (n != 31);
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**SUB (vector)**

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Asm** `<V><d>`, `<V><n>`, `<V><m>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');

### Vector

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 1   | 0   | 1   | 1   | 1   | 0   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size|     |     |     |     |     |     | Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|     | U   |     |     |     |     |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Asm** `<Vd><T>`, `<Vn><T>`, `<Vm><T>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the “Rd” field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
b bits(esize) element1;
b bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see RSUBHN.

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUDOT (by element)**

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an **OPTIONAL** instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

**Vector (FEAT_I8MM)**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| **US** |

**SUDOT** `<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]`

```java
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in "Q":
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in "Q":
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- `<index>` Is the immediate index of a quadtuplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.
Operation

```
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
  bits(32) res = Elem[operand3, e, 32];
  for b = 0 to 3
    integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
    integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
    res = res + element1 * element2;
  Elem[result, e, 32] = res;
V[d] = result;
```
**SUQADD**

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |

**SUQADD** `<V>`<d>, `<V>`<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |

**SUQADD** `<Vd>`.<T>, `<Vn>`.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<n>` Is the number of the SIMD&FP source register, encoded in the "Rn" field.

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;
for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values. The SXTL instruction extracts the source vector from the lower half of the source register. The SXTL2 instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSHLL, SSHLL2. This means:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | Rn | Rd |

U immh immb

SXTL(2) <Vd>, <Ta>, <Vin>, <Tb>

is equivalent to

SSHLL(2) <Vd>, <Ta>, <Vin>, <Tb>, #0

and is the preferred disassembly when BitCount(immh) == 1.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vin> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
**Operation**

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the \texttt{CPACR\_EL1, CPTR\_EL2,} and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
op & Rd & len & Rn & Vm & Vd & Ta & \hline
\end{tabular}
\end{table}

Two register table (len == 01)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');

\textbf{Assembler Symbols}

- \texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Ta>} Is an arrangement specifier, encoded in "Q":
  \begin{table}[h]
  \centering
  \begin{tabular}{|c|c|}
  \hline
  Q & <Ta> \\
  \hline
  0 & 8B \\
  1 & 16B \\
  \end{tabular}
  \end{table}
- \texttt{<Vn>} For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.
  For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.
- \texttt{<Vn+1>} Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.
- \texttt{<Vn+2>} Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.
- \texttt{<Vn+3>} Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.
- \texttt{<Vm>} Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Two register table (len == 01)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>

Three register table (len == 10)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>

Four register table (len == 11)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>

Single register table (len == 00)

TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);
type m = UInt(Rm);

type data_size = if Q == '1' then 128 else 64;
type elements = data_size DIV 8;
type regs = UInt(len) + 1;
bool is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**TRN1**

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

**Note**

By using this instruction with TRN2, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

![Diagram showing the operation of TRN1 and TRN2](image)

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

**Note**

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

![TRN1 example](image)

![TRN2 example](image)

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 | 0 | 1 | 1 | 0 | size | 0 | Rm | 0 | 1 | 1 | 0 | 1 | 0 | Rn | Rd | op
```

TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;
result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register. The UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>U</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>1</td>
</tr>
<tr>
<td>Rd</td>
<td>Rn</td>
</tr>
</tbody>
</table>
```

UABAL{2} <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

| 2 | Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td>Rn</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

UABD `<Vd>`..<T>, `<Vn>`..<T>, `<Vm>`..<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  absdiff = Abs(element1-element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UABDL, UABDL2**

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register. The UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>**

Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>**

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Tb>**

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vm>**

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

UADALP <Vd>.<Ta>, <Vn>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(data_size) operand = V[n];
bits(data_size) result;

bits(2*esize) sum;
if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register. The UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\text{CheckFPAdvSIMDEnabled64}();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;
sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);
V[d] = sum<2*esize-1:0>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register. The UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| O | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 0 | 1 | 0 | 0 | Rn | Rd |

UAD\texttt{D\{2\} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
\end{verbatim}

\textbf{Assembler Symbols}

\textbf{2} Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

\begin{verbatim}
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textbf{<Ta>} Is an arrangement specifier, encoded in "size":

\begin{verbatim}
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\textbf{<Vm>} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

\textbf{<Tb>} Is an arrangement specifier, encoded in "size:Q":

\begin{verbatim}
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
\end{verbatim}
Operation

\textbf{CheckFPAdvSIMDEnabled64}();

\texttt{bits(2*datasize) operand1 = V[n];}
\texttt{bits(datasize) operand2 = Vpart[m, part];}
\texttt{bits(2*datasize) result;}
\texttt{integer element1;}
\texttt{integer element2;}
\texttt{integer sum;}

\texttt{for e = 0 to elements-1}
\texttt{element1 = \texttt{Int(}Elem\texttt{(operand1, e, 2*esize), unsigned);} \texttt{if sub_op then}
\texttt{element2 = \texttt{Int(}Elem\texttt{(operand2, e, esize), unsigned);}}
\texttt{element2 = \texttt{Int(}Elem\texttt{(operand2, e, esize), unsigned);}}
\texttt{if sub_op then}
\texttt{sum = element1 - element2;}
\texttt{else}
\texttt{sum = element1 + element2;}
\texttt{Elem[result, e, 2*esize] = sum<2*esize-1:0>;}\texttt{V[d] = result;}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UCVTF (scalar, fixed-point)**

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**32-bit to half-precision (sf == 0 & ftype == 11) (FEAT_FP16)**

UCVTF <Hd>, <Wn>, #<fbits>

**32-bit to single-precision (sf == 0 & ftype == 00)**

UCVTF <Sd>, <Wn>, #<fbits>

**32-bit to double-precision (sf == 0 & ftype == 01)**

UCVTF <Dd>, <Wn>, #<fbits>

**64-bit to half-precision (sf == 1 & ftype == 11) (FEAT_FP16)**

UCVTF <Hd>, <Xn>, #<fbits>

**64-bit to single-precision (sf == 1 & ftype == 00)**

UCVTF <Sd>, <Xn>, #<fbits>

**64-bit to double-precision (sf == 1 & ftype == 01)**

UCVTF <Dd>, <Xn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPRounding rounding;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end_case;

if sf == '0' & scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[]);
Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```java
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bites(fsize) fltval;
bites(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, TRUE, fpcr, rounding);
V[d] = fltval;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the **FPCR**, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 32-bit to half-precision (sf == 0 && ftype == 11) (FEAT_FP16)

UCVTF <Hd>, <Wn>

### 32-bit to single-precision (sf == 0 && ftype == 00)

UCVTF <Sd>, <Wn>

### 32-bit to double-precision (sf == 0 && ftype == 01)

UCVTF <Dd>, <Wn>

### 64-bit to half-precision (sf == 1 && ftype == 11) (FEAT_FP16)

UCVTF <Hd>, <Xn>

### 64-bit to single-precision (sf == 1 && ftype == 00)

UCVTF <Sd>, <Xn>

### 64-bit to double-precision (sf == 1 && ftype == 01)

UCVTF <Dd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPRoundingMode(FPCR[]);
Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
FPCRTyperefcr = FPCR[];
boolean merge = IsMerging(refcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[filtval, 0, fltsize] = FixedToFP(intval, 0, TRUE, refcr, rounding);
V[d] = fltval;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |

UCVTF <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

(8 bits)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |

UCVTF <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector half precision

(8 bits)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

**Vector single-precision and double-precision**

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>Vd</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<

---

<table>
<thead>
<tr>
<th>Vd</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---

<table>
<thead>
<tr>
<th>Hn</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---

<table>
<thead>
<tr>
<th>V</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---

<table>
<thead>
<tr>
<th>d</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---

<table>
<thead>
<tr>
<th>n</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---

<table>
<thead>
<tr>
<th>V</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---

<table>
<thead>
<tr>
<th>U</th>
<th>T</th>
<th>Vn</th>
</tr>
</thead>
</table>

---

0 | 1 |

---
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

FPCRTypen fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

FPRoundingn rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding);
V[d] = result;
UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an **OPTIONAL** instruction. From Armv8.4 it is mandatory for all implementations to support it.

*Note*

**ID_AA64ISAR0_EL1**.DP indicates whether this instruction is supported.

### Vector

**FEAT_DotProd**

<table>
<thead>
<tr>
<th>U</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UDOT <Vd>, <Ta>, <Vn>, <Tb>, <Vm>.4B[index]

```plaintext
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- **<Ta>** Is an arrangement specifier, encoded in “Q”:
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>** Is an arrangement specifier, encoded in “Q”:
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- **<index>** Is the element index, encoded in the "H:L" fields.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

**Note**

`ID_AA64ISAR0_EL1`.DP indicates whether this instruction is supported.

**Vector**

(FEAT DotProd)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | size | 0  |   | Rm | 1  | 0  | 0  | 1  | 0  | 1  |   | Rn |   | Rd |

UDOT `<Vd>..<Ta>..<Vn>..<Tb>..<Vm>`

if `!HaveDOTPExt()` then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

`<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

`CheckFPAdvSIMDEnabled64();`

`bits(datasize) operand1 = V[n];`
`bits(datasize) operand2 = V[m];`
`bits(datasize) result;`

result = V[d];
for e = 0 to elements-1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
    else
      element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;`
**UHADD**

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see **URHADD**.

Depending on the settings in the **CPACR_EL1, CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

- integer d = **UInt**(Rd);
- integer n = **UInt**(Rn);
- integer m = **UInt**(Rm);
- if size == '11' then UNDEFINED;
- integer esize = 8 << **UInt**(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm | 0  | 0  | 1  | 0  | 0  | 1  | Rn | 0  | 0  | 0  | 1  | Rd |

UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UMAX**

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|------------------------|--------|------------------------|
| 0 | O | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 |
| Rd | Rm | 0 | 1 | 1 | 0 | 0 | 1 |
|   | Rn |
```

**Asm**

```
UMAX <Vd>..<T>, <Vn>..<T>, <Vm>..<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler Symbols**

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in “size:Q”:
  ```
  size   Q   <T>
  00     0   8B
  00     1   16B
  01     0   4H
  01     1   8H
  10     0   2S
  10     1   4S
  11     x   RESERVED
  ```
- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bites(datasize) operand1 = V[n];
bites(datasize) operand2 = V[m];
bites(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler Symbols**

~<Vd>.<T>, <Vn>.<T>, <Vm>.<T>

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();

\[
\text{bits(datasize) operand1} = V[n]; \]
\[
\text{bits(datasize) operand2} = V[m]; \]
\[
\text{bits(datasize) result}; \]
\[
\text{bits(2*datasize) concat} = \text{operand2:operand1}; \]
\[
\text{integer element1}; \]
\[
\text{integer element2}; \]
\[
\text{integer maxmin}; \]

for e = 0 to elements-1

\[
\text{element1} = \text{Int(Elem[concat, 2*e, esize], unsigned}); \]
\[
\text{element2} = \text{Int(Elem[concat, (2*e)+1, esize], unsigned}); \]
\[
\text{maxmin} = \text{if minimum then Min(element1, element2) else Max(element1, element2)}; \]
\[
\text{Eelem[result, e, esize] = maxmin<esize-1:0>}; \]

\[
V[d] = \text{result}; \]
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMAXV**

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**UMAXV <V><d>, <Vn>.<T>**

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(\text{size:Q} == '100'\) then UNDEFINED;
if \(\text{size} == '11'\) then UNDEFINED;
integer esize = 8 << \(\text{UInt}(\text{size})\);
integer datasize = if \(\text{Q} == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (\(U == '1'\));
boolean min = (\(\text{op} == '1'\));

**Assembler Symbols**

\(<V>\) Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = \(\text{Int}(\text{Elem}[\text{operand}, \text{esize}, \text{unsigned}])\);
for e = 1 to elements-1" element = \(\text{Int}(\text{Elem}[\text{operand}, e, \text{esize}], \text{unsigned})\);" maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

UMAXV
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMIN**

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UMIN** <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result; integer element1; integer element2; integer maxmin;
for e = 0 to elements-1 
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
| 0 | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm    | 1 | 0 | 1 | 0 | 1 | 1 | Rd    |
\end{verbatim}

\textbf{Assembler Symbols}

\textbf{<Vd> } Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textbf{<T> } Is an arrangement specifier, encoded in "size:Q":

\begin{verbatim}
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{<Vn> } Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\textbf{<Vm> } Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

\textbf{Operation}

\begin{verbatim}
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = \texttt{Int}(Elem[concat, 2*e, esize], unsigned);
    element2 = \texttt{Int}(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then \texttt{Min}(element1, element2) else \texttt{Max}(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 | O | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
```

**UMINV `<V><d>`, `<Vn>`. `<T>`**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size = '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

**Assembler Symbols**

- `<V>`: Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>`: Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
\begin{verbatim}
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
\end{verbatim}
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

\[
\begin{array}{c|c}
  Q & 2 \\
  \hline
  0 & \text{[absent]} \\
  1 & \text{[present]} \\
\end{array}
\]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

\[
\begin{array}{c|c}
  \text{size} & \text{<Ta>} \\
  \hline
  00 & \text{RESERVED} \\
  01 & 4S \\
  10 & 2D \\
  11 & \text{RESERVED} \\
\end{array}
\]

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Hexadecimal representation of the UMLAL instruction](image)

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>absent</td>
</tr>
<tr>
<td>1</td>
<td>present</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLSL instruction extracts vector elements from the lower half of the first source register. The UMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U   | O  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | H  | 0  | Rn | Rd |
|     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

UMLSL2 \{2\}  \(<Vd>\).\(<Ta>\),  \(<Vn>\).\(<Tb>\),  \(<Vm>\).\(<Ts>\)\[[<index>]\]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”: 

UMLSL, UMLSL2 (by element)
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bv(2*datasize) result;
in integer element1;
in integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMLSL` instruction extracts each source vector from the lower half of each source register. The `UMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>UMLSL2</th>
<th>&lt;Vd&gt;, &lt;Ta&gt;, &lt;Vn&gt;, &lt;Vm&gt;.&lt;Tb&gt;</th>
</tr>
</thead>
</table>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

#### Assembler Symbols

- **2**: Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

- **<Vd>**: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Ta>**: Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>**: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Tb>**: Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vm>**: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an **OPTIONAL** instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. `ID_AA64ISAR1_EL1.I8MM` indicates whether this instruction is supported.

**Vector**

(\texttt{FEAT\_I8MM})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| B  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\texttt{UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B}

if \texttt{!HaveInt8MatMulExt()} then UNDEFINED;
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
integer d = \texttt{UInt}(Rd);

**Assembler Symbols**

- \texttt{<Vd>} is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- \texttt{<Vn>} is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Vm>} is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(128) operand1 = V[n];}
\texttt{bits(128) operand2 = V[m];}
\texttt{bits(128) addend = V[d];}

\texttt{V[d] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);}
Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general).

32-bit (Q == 0)

UMOV <Wd>, <Vn>.<Ts>[<index>]

64-bit (Q == 1 && imm5 == x1000)

UMOV <Xd>, <Vn>.<Ts>[<index>]

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxxx1' size = 0;    // UMOV Wd, Vn.B
    when '0xxx10' size = 1;    // UMOV Wd, Vn.H
    when '0xx100' size = 2;    // UMOV Wd, Vn.S
    when '1x1000' size = 3;    // UMOV Xd, Vn.D
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Ts> For the 32-bit variant: is an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is an element size specifier, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xx100</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the 32-bit variant: is the element index encoded in "imm5":

UMOV
### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to general)</td>
<td>imm5 == ‘x1000’</td>
</tr>
<tr>
<td>MOV (to general)</td>
<td>imm5 == ‘xx100’</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxsize) operand = V[n];

X[d] = ZeroExtend(Elem[operand, index, esize], datasize);
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register. The UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 1 size L M Rm 1 0 1 0 H 0 Rn Rd U

UMULL2[2] <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

2            Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>            Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>            Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>            Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>            Is an arrangement specifier, encoded in “size:Q”:
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register. The UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 1 | 0 | 0 | 0 | Rn | Rd |
| U |
```

UMULL(2) <Vd>, <Ta>, <Vn>, <Vm>, <Vm>, <Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
in...
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-------------------|-------------------|
| 0 1 1 1 1 1 1 0 | size | 1 | Rm | 0 0 0 0 1 1 | Rn | Rd |
| U                 |
```

UQADD <V>d>, <V>n>, <V>m>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- integer esize = 8 << UInt(size);
- integer datasize = esize;
- integer elements = 1;
- boolean unsigned = (U == '1');

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-------------------|-------------------|
| 0 | Q | 1 0 1 1 1 0 | size | 1 | Rm | 0 0 0 0 1 1 | Rn | Rd |
| U                 |
```

UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size:Q == '110' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');

Assembler Symbols

- <V> Is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <d> Is the number of the SIMD&FP destination register, in the "Rd" field.
- <n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- <m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
ineger element2;
ineger sum;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
element2 = Int(Elem[operand2, e, esize], unsigned);
sum = element1 + element2;
(Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see UQSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0   1   1   1   1   1   1   0 | size | 1 | Rm | 0   1   0   1   1   1 | Rn | Rd |
| U   | R   | S |

UQRSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0   Q   1   0   1   1   1   0 | size | 1 | Rm | 0   1   0   1   1   1 | Rn | Rd |
| U   | R   | S |

UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see \texttt{UQRSHRN}.

The \texttt{UQRSHRN} instruction writes the vector to the lower half of the destination register and clears the upper half, while the \texttt{UQRSHRN2} instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \texttt{FPSR.QC} is set.

Depending on the settings in the \texttt{CPACR_EL1}, \texttt{CPTR_EL2}, and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}

\textbf{Scalar}

\begin{verbatim}
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| Rn | Rd | op | immh | U | shift |
|-----------------|-----------------|-----------------|
\end{verbatim}

\texttt{UQRSHRN <Vb><d>, <Va>n>, #<shift>}

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << \texttt{HighestSetBit}(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - \texttt{UInt}(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

\textbf{Vector}

\begin{verbatim}
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| Rn | Rd | op | immh | U | shift |
|-----------------|-----------------|-----------------|
\end{verbatim}

\texttt{UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>}

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

if immh == '0000' then \texttt{SEE(asimdimm)};
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << \texttt{HighestSetBit}(immh);
integer datasize = 64;
integer part = \texttt{UInt}(Q);
integer elements = datasize \texttt{DIV} esize;

integer shift = (2 * esize) - \texttt{UInt}(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
### Assembler Symbols

2  

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb>  

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>  

Is an arrangement specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>  

Is the destination width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>  

Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va>  

Is the source width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>  

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift>  

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td><strong>SEF Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 != 0000 immh 0 1 1 1 0 1 Rn Rd

UQSHL <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 != 0000 immh 0 1 1 1 0 1 Rn Rd

UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```
UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 | size | 1 | Rm | 0 1 0 | 0 1 1 | Rn | Rd

UQSHL <V<d>, <V<n>, <V<m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 0 | size | 1 | Rm | 0 1 0 | 0 1 1 | Rn | Rd

UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN.

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!= 0000</td>
<td>immh</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>immh</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>!= 0000</td>
<td>immh</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>immh</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQSHRN(2) <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SFE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \texttt{FPSR.QC} is set.

Depending on the settings in the \texttt{CPACR_EL1}, \texttt{CPTR_EL2}, and \texttt{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{Scalar} and \texttt{Vector}

**Scalar**

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & \text{size} & 1 & \text{Rm} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rn} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rd} & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

UQSUB \texttt{<V><d>, <V><n>, <V><m>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 \ll UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U \== '1');

**Vector**

\[
\begin{array}{cccccccccccccccccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & \text{Rm} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rn} & 0 & 0 & 1 & 0 & 1 & 1 & \text{Rd} & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

UQSUB \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 \ll UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U \== '1');

**Assembler Symbols**

\texttt{<V>} Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\texttt{<d>} Is the number of the SIMD&FP destination register, in the "Rd" field.

\texttt{<n>} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<m>} Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<br>

<vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem(operand1, e, esize], unsigned);
    element2 = Int(Elem(operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem(result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQXTN, UQXTN2**

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

The **UQXTN** instruction writes the vector to the lower half of the destination register and clears the upper half, while the **UQXTN2** instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 1 1 1 1 1 0     | size 1 0 0 0 0    | 1 0 1 0 0 1 0     | Rn                | Rd                |
```

**UQXTN** <Vb><d>, <Va><n>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

**Vector**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 |Q 1| 0 1 1 1 0     | size 1 0 0 0 0    | 1 0 1 0 0 1 0     | Rn                | Rd                |
```

**UQXTN(2)** <Vd>.<Tb>, <Vn>.<Ta>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

**Assembler Symbols**

```
2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Tb>  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>REServered</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta>  Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<Vb>  Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va>  Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
  element = Elem[operand, e, 2*esize];
  (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
  if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URECPE**

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 0  | 1  | 1  | 0  | 1  | | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

**URECPE** `<Vd>.<T>, <Vn>.<T>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;
for e = 0 to elements-1
   element = Elem[operand, e, 32];
   Elem[result, e, 32] = UnsignedRecipEstimate(element);
V[d] = result;
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see UHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  Elem[result, e, esize] = (element1+element2+1)<esize:1>;
V[d] = result;
**URSHL**

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |   | size | 1  | Rm | 0  | 1  | 0  | 1  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |
```

**URSHL** `<V><d>, <V><n>, <V><m>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

**Vector**

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U | 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  |   | size | 1  | Rm | 0  | 1  | 0  | 1  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |
```

**URSHL** `<Vd>.<T>, <Vn>.<T>, <Vm>.<T>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see URSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 != 0000 immb 0 0 1 0 0 1 Rn Rd

URSHR <V>d>, <V>n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 1 0 != 0000 immb 0 0 1 0 0 1 Rn Rd

URSHR <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>D</td>
</tr>
</tbody>
</table>
Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSQRTE**

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |

**Asm** URSQRTE `<Vd>`.<T>, `<Vn>`.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in "sz:Q":

```
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(32) element;
for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 1  | 1  | 1  | 1  | 1  | 0  | != | 0000| immb| 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U | immh| o1 | o0 |

**URSRA <V>d>, <V>cn>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Vector**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | != | 0000| immb| 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U | immh| o1 | o0 |

**URSRA <V>d>.<T>, <V>..<T>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh = '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler Symbols**

<

Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
  element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an **OPTIONAL** instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. **ID_AA64ISAR1_EL1**.I8MM indicates whether this instruction is supported.

Vector
(**FEAT_I8MM**)  

```
0 | Q | 0 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 1 | 1 | 1 | H | 0 | Rn | Rd
```

**USDOT** `<Vd>`, `<Ta>`, `<Vn>`, `<Tb>`, `<Vm>`.4B[<index>]

if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');

integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

Assembler Symbols

**<Vd>** is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

**<Ta>** is an arrangement specifier, encoded in "Q":

```
Q | <Ta>
---|---
0 | 25
1 | 45
```

**<Vn>** is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Tb>** is an arrangement specifier, encoded in "Q":

```
Q | <Tb>
---|---
0 | 8B
1 | 16B
```

**<Vm>** is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

**<index>** is the immediate index of a quaduplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{bits} \text{(datasize)} \text{ operand1} = V[n];
\]

\[
\text{bits}(128) \text{ operand2} = V[m];
\]

\[
\text{bits} \text{(datasize)} \text{ operand3} = V[d];
\]

\[
\text{bits} \text{(datasize)} \text{ result};
\]

for \( e = 0 \) to \( \text{elements} - 1 \)

\[
\text{bits}(32) \text{ res} = \text{Elem}[\text{operand3}, e, 32];
\]

for \( b = 0 \) to \( 3 \)

\[
\text{integer element1} = \text{Int} \text{(Elem}[\text{operand1}, 4*\text{e}+\text{b}, 8], \text{op1} \_\text{unsigned});
\]

\[
\text{integer element2} = \text{Int} \text{(Elem}[\text{operand2}, 4*\text{i}+\text{b}, 8], \text{op2} \_\text{unsigned});
\]

\[
\text{res} = \text{res} + \text{element1} \times \text{element2};
\]

\[
\text{Elem}[\text{result}, e, 32] = \text{res};
\]

\[
V[d] = \text{result};
\]
USDOT (vector)

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. \textit{ID_AA64ISAR1_EL1}.I8MM indicates whether this instruction is supported.

Vector (FEAT_I8MM)

![Table](table.png)

**USDOT \(<V_d>., <V_n>., <V_m>., <T_a>, <T_b>, <T_m>\)**

```plaintext
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

**Assembler Symbols**

- \(<V_d>\) Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- \(<T_a>\) Is an arrangement specifier, encoded in "Q":
  - \(Q \quad <T_a>\)
    - 0 25
    - 1 45
- \(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<T_b>\) Is an arrangement specifier, encoded in "Q":
  - \(Q \quad <T_b>\)
    - 0 8B
    - 1 16B
- \(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \(V[n]\);
bits(datasize) operand2 = \(V[m]\);
bits(datasize) operand3 = \(V[d]\);
bits(datasize) result;
for e = 0 to elements-1
  bits(32) res = \(\text{Elem}[operand3, e, 32]\);
  for b = 0 to 3
    integer element1 = UInt(\(\text{Elem}[operand1, 4*e+b, 8]\));
    integer element2 = SInt(\(\text{Elem}[operand2, 4*e+b, 8]\));
    res = res + element1 * element2;
  \(\text{Elem}[result, e, 32] = res;\)
\(V[d] = result;\)
```

USDOT (vector)
USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see URSYL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|------------------|------------------|
| 0 1 1 1 1 1 1 0 | size 1           | Rm 0 1 0 0 0 1   | Rn | Rd |
|                 |                  | U               |     |     |

USHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|------------------|------------------|
| 0 0 Q 1 0 1 1 1 0 | size 1           | Rm 0 1 0 0 0 1   | Rn | Rd |
|                 |                  | U               |     |     |

USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    else
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';

    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USHLL, USHLL2**

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register. The USHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the `CPACR_EL1, CPTR_EL2, and CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `UXTL, UXTL2`.

**Assembler Symbols**

<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>[absent]</th>
<th>1</th>
<th>[present]</th>
</tr>
</thead>
</table>

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>** Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Tb>** Is an arrangement specifier, encoded in “immh:Q”:

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```
<table>
<thead>
<tr>
<th>immh</th>
<th>(Q)</th>
<th>&lt;(Tb)&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>08B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{shift}>\) Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;(\text{shift})&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>UInt(immh:immb)-8</td>
</tr>
<tr>
<td>001x</td>
<td>UInt(immh:immb)-16</td>
</tr>
<tr>
<td>01xx</td>
<td>UInt(immh:immb)-32</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTL, UXTL2</td>
<td>immh == '000' &amp;&amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USHR**

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see **URSHR**.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

U  
immh  
o1 o0

USHR **<V><d>, <V><n>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

U  
immh  
o1 o0

USHR **<Vd>..<T>, <Vn>..<T>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “immh”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>immh</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an optional instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. \textit{ID\_AA64ISAR1\_EL1}\_I8MM indicates whether this instruction is supported.

Vector
\textit{(FEAT\_I8MM)}

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Rm} \hspace{1cm} \textbf{Rn} \hspace{1cm} \textbf{Rd}

\textbf{USMMLA} \textless \textbf{Vd}\textgreater\.4S, \textless \textbf{Vn}\textgreater\.16B, \textless \textbf{Vm}\textgreater\.16B

\textbf{Assembler Symbols}

\textbf{<Vd>} \hspace{1cm} \textbf{Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.}

\textbf{<Vn>} \hspace{1cm} \textbf{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.}

\textbf{<Vm>} \hspace{1cm} \textbf{Is the name of the second SIMD&FP source register, encoded in the "Rm" field.}

\textbf{Operation}

\textbf{CheckFPAdvSIMDEnabled64}();

\textbf{bits}(128) \textbf{operand1} = \textbf{V}[n];

\textbf{bits}(128) \textbf{operand2} = \textbf{V}[m];

\textbf{bits}(128) \textbf{addend} = \textbf{V}[d];

\textbf{V}[d] = \textbf{MatMulAdd}(\textbf{addend}, \textbf{operand1}, \textbf{operand2}, \text{TRUE, FALSE});
USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 size 1 0 0 0 0 0 0 1 1 1 0 Rn  Rd

USQADD <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 0 size 1 0 0 0 0 0 0 1 1 1 0 Rn  Rd

USQADD <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>00</th>
<th>01</th>
<th>01</th>
<th>10</th>
<th>10</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>8B</td>
<td>16B</td>
<td>4H</td>
<td>8H</td>
<td>2S</td>
<td>4S</td>
<td>RESERVED</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
  op1 = Int(Elem[operand, e, esize], !unsigned);
  op2 = Int(Elem[operand2, e, esize], unsigned);
  (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
  if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 != 0000 immb 0 0 0 1 0 1 Rn Rd
U immh 01 00
```

USRA <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 1 1 1 1 0 != 0000 immb 0 0 0 1 0 1 Rn Rd
U immh 01 00
```

USRA <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '0000' then SEE(asimdimm);
if imm<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>
Is the number of the SIMD&FP destination register, in the "Rd" field.

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**USUBL, USUBL2**

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register. The USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | O | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 0 | 0 | 0 | Rn | Rd | U | o1 |

**USUBL2**<Vd>,<Ta>,<Vn>,<Vm>,<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
in
integer element2;
in
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USUBW, USUBW2**

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are unsigned integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register. The USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 1 | 0 | Rn | Rd | 0 | 1
U
```

**USUBW{2}** <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

1. integer d = UInt(Rd);
2. integer n = UInt(Rn);
3. integer m = UInt(Rm);
4. if size == '11' then UNDEFINED;
5. integer esize = 8 << UInt(size);
6. integer datasize = 64;
7. integer part = UInt(Q);
8. integer elements = datasize DIV esize;
9. boolean sub_op = (o1 == '1');
10. boolean unsigned = (U == '1');

**Assembler Symbols**

2. Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UXTL, UXTL2**

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register. The UXTL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of USHLL, USHLL2. This means:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

```
0  Q 1 0 1 1 1 0 | != 0000 0 0 0 1 0 1 0 0 1 | Rd
U          immh  immb
```

**UXTL(2) <Vd>.<Ta>, <Vn>.<Tb>**

is equivalent to

**USHLL(2) <Vd>.<Ta>, <Vn>.<Tb>, #0**

and is the preferred disassembly when BitCount(immh) == 1.

**Assembler Symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
Q   2
0   [absent]
1   [present]
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

```
immh  <Ta>
0000  SEE Advanced SIMD modified immediate
0001  8H
001x  4S
01xx  2D
1xxx  RESERVED
```

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

```
immh  Q   <Tb>
0000  x  SEE Advanced SIMD modified immediate
0001  0  8B
0001  1  16B
001x  0  4H
001x  1  8H
01xx  0  2S
01xx  1  4S
1xxx  x  RESERVED
```
Operation

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UZP1

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction can be used with UZP2 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UZP2**

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

**Note**

This instruction can be used with **UZP1** to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.

![Figure showing the operation of UZP1 and UZP2]

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

**Assembler Symbols**

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operandl = \text{V}[n];
bits(datasize) operandh = \text{V}[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    \text{Elem}[result, e, esize] = \text{Elem}[zipped, 2*e+part, esize];
\text{V}[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when `FEAT_SHA3` is implemented.

**Advanced SIMD**

`(FEAT_SHA3)`

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

---

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 1 1 1 0 | size 1 0 0 0 0 | Rd
0 0 0 1 1 | Rn
```

XTN{2} <Vd>,<Tb>, <Vn>,<Ta>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

```
size 0 1 8B 16B 4H 8H 2S 4S x RESERVED
```

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

```
size 8H 4S 2D x RESERVED
```
Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP
registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination
SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with
subsequent pairs taken alternately from each source register.

Note

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
Vn | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0
Vm | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0

ZIP1, doubleword

Vd | B3 | A3 | B2 | A2 | B1 | A1 | B0 | A0

ZIP2, doubleword

Vd | B7 | A7 | B6 | A6 | B5 | A5 | B4 | A4
```

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

```
0 Q 0 0 1 1 1 0 size 0 Rm 0 0 1 1 1 0 Rn Rd
```

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
  Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
  Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

**Note**

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
Vn | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
Vm | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
```

ZIP1.8, doubleword

```
Vd | B3 | A2 | B2 | A2 | B1 | A1 | B0 | A0 |
```

ZIP2.8, doubleword

```
Vd | B7 | A7 | B6 | A6 | B5 | A5 | B4 | A4 |
```

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0  Q  0  0  1  1  1  0  size  0  Rd  0  1  1  1  1  0  Rn  Q  0  0  1  1  1  0  size  0  Rd
op       Q
0  Q  0  0  1  1  1  0  size  0  Rd  0  1  1  1  1  0  Rn  Q  0  0  1  1  1  0  size  0  Rd
```

ZIP2 `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<T>`

```java
integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
A64 -- SVE Instructions (alphabetic order)

**ABS**: Absolute value (predicated).

**ADD (immediate)**: Add immediate (unpredicated).

**ADD (vectors, predicated)**: Add vectors (predicated).

**ADD (vectors, unpredicated)**: Add vectors (unpredicated).

**ADDPL**: Add multiple of predicate register size to scalar register.

**ADDVL**: Add multiple of vector register size to scalar register.

**ADR**: Compute vector address.

**AND (immediate)**: Bitwise AND with immediate (unpredicated).

**AND (predicates)**: Bitwise AND predicates.

**AND (vectors, predicated)**: Bitwise AND vectors (predicated).

**AND (vectors, unpredicated)**: Bitwise AND vectors (unpredicated).

**ANDS**: Bitwise AND predicates, setting the condition flags.

**ANDV**: Bitwise AND reduction to scalar.

**ASR (immediate, predicated)**: Arithmetic shift right by immediate (predicated).

**ASR (immediate, unpredicated)**: Arithmetic shift right by immediate (unpredicated).

**ASR (vectors)**: Arithmetic shift right by vector (predicated).

**ASR (wide elements, predicated)**: Arithmetic shift right by 64-bit wide elements (predicated).

**ASR (wide elements, unpredicated)**: Arithmetic shift right by 64-bit wide elements (unpredicated).

**ASRD**: Arithmetic shift right for divide by immediate (predicated).

**ASRR**: Reversed arithmetic shift right by vector (predicated).

**BFCVT**: Floating-point down convert to BFloat16 format (predicated).

**BFCVTNT**: Floating-point down convert and narrow to BFloat16 (top, predicated).

**BFDOT (indexed)**: BFloat16 floating-point indexed dot product.

**BFDOT (vectors)**: BFloat16 floating-point dot product.

**BFMLALB (indexed)**: BFloat16 floating-point multiply-add long to single-precision (bottom, indexed).

**BFMLALB (vectors)**: BFloat16 floating-point multiply-add long to single-precision (bottom).

**BFMLALT (indexed)**: BFloat16 floating-point multiply-add long to single-precision (top, indexed).

**BFMLALT (vectors)**: BFloat16 floating-point multiply-add long to single-precision (top).

**BFMMLA**: BFloat16 floating-point matrix multiply-accumulate.

**BIC (immediate)**: Bitwise clear bits using immediate (unpredicated): an alias of AND (immediate).

**BIC (predicates)**: Bitwise clear predicates.

**BIC (vectors, predicated)**: Bitwise clear vectors (predicated).

**BIC (vectors, unpredicated)**: Bitwise clear vectors (unpredicated).

**BICS**: Bitwise clear predicates, setting the condition flags.
**BRKA**: Break after first true condition.
**BRKAS**: Break after first true condition, setting the condition flags.
**BRKB**: Break before first true condition.
**BRKBS**: Break before first true condition, setting the condition flags.
**BRKN**: Propagate break to next partition.
**BRKNS**: Propagate break to next partition, setting the condition flags.
**BRKPA**: Break after first true condition, propagating from previous partition.
**BRKPAS**: Break after first true condition, propagating from previous partition and setting the condition flags.
**BRKPB**: Break before first true condition, propagating from previous partition.
**BRKPBS**: Break before first true condition, propagating from previous partition and setting the condition flags.

**CLASTA** (scalar): Conditionally extract element after last to general-purpose register.
**CLASTA** (SIMD&FP scalar): Conditionally extract element after last to SIMD&FP scalar register.
**CLASTA** (vectors): Conditionally extract element after last to vector register.
**CLASTB** (scalar): Conditionally extract last element to general-purpose register.
**CLASTB** (SIMD&FP scalar): Conditionally extract last element to SIMD&FP scalar register.
**CLASTB** (vectors): Conditionally extract last element to vector register.

**CLS**: Count leading sign bits (predicated).
**CLZ**: Count leading zero bits (predicated).

**CMP<cc>** (immediate): Compare vector to immediate.
**CMP<cc>** (vectors): Compare vectors.
**CMP<cc>** (wide elements): Compare vector to 64-bit wide elements.

**CMPLT** (vectors): Compare signed less than vector, setting the condition flags: an alias of CMP<cc> (vectors).

**CMPLE** (vectors): Compare signed less than or equal to vector, setting the condition flags: an alias of CMP<cc> (vectors).

**CMPLS** (vectors): Compare unsigned lower or same as vector, setting the condition flags: an alias of CMP<cc> (vectors).

**CMPLT** (vectors): Compare signed less than vector, setting the condition flags: an alias of CMP<cc> (vectors).

**CNOT**: Logically invert boolean condition in vector (predicated).

**CNT**: Count non-zero bits (predicated).
**CNTB, CNTD, CNTH, CNTW**: Set scalar to multiple of predicate constraint element count.
**CNTP**: Set scalar to count of true predicate elements.

**COMPACT**: Shuffle active elements of vector to the right and fill with zero.

**CPY** (immediate, merging): Copy signed integer immediate to vector elements (merging).
**CPY** (immediate, zeroing): Copy signed integer immediate to vector elements (zeroing).
**CPY** (scalar): Copy general-purpose register to vector elements (predicated).
**CPY** (SIMD&FP scalar): Copy SIMD&FP scalar register to vector elements (predicated).

**CTERMEQ, CTERMNE**: Compare and terminate loop.
DECB, DECD, DECH, DECW (scalar): Decrement scalar by multiple of predicate constraint element count.

DECD, DECH, DECW (vector): Decrement vector by multiple of predicate constraint element count.

DECP (scalar): Decrement scalar by count of true predicate elements.

DECP (vector): Decrement vector by count of true predicate elements.

DUP (immediate): Broadcast signed immediate to vector elements (unpredicated).

DUP (indexed): Broadcast indexed element to vector (unpredicated).

DUP (scalar): Broadcast general-purpose register to vector elements (unpredicated).

DUPM: Broadcast logical bitmask immediate to vector (unpredicated).

EON: Bitwise exclusive OR with inverted immediate (unpredicated): an alias of EOR (immediate).

EOR (immediate): Bitwise exclusive OR with immediate (unpredicated).

EOR (predicates): Bitwise exclusive OR predicates.

EOR (vectors, predicated): Bitwise exclusive OR vectors (predicated).

EOR (vectors, unpredicated): Bitwise exclusive OR vectors (unpredicated).

EORS: Bitwise exclusive OR predicates, setting the condition flags.

EORV: Bitwise exclusive OR reduction to scalar.

EXT: Extract vector from pair of vectors.

FABD: Floating-point absolute difference (predicated).

FABS: Floating-point absolute value (predicated).

FAC<cc>: Floating-point absolute compare vectors.

FACLE: Floating-point absolute compare less than or equal: an alias of FAC<cc>.

FACLT: Floating-point absolute compare less than: an alias of FAC<cc>.

FADD (immediate): Floating-point add immediate (predicated).

FADD (vectors, predicated): Floating-point add vector (predicated).

FADD (vectors, unpredicated): Floating-point add vector (unpredicated).

FADDA: Floating-point add strictly-ordered reduction, accumulating in scalar.

FADDV: Floating-point add recursive reduction to scalar.

FCADD: Floating-point complex add with rotate (predicated).

FCM<cc> (vectors): Floating-point compare vectors.

FCM<cc> (zero): Floating-point compare vector with zero.

FCMLA (indexed): Floating-point complex multiply-add by indexed values with rotate.

FCMLA (vectors): Floating-point complex multiply-add with rotate (predicated).

FCMLE (vectors): Floating-point compare less than or equal to vector: an alias of FCM<cc> (vectors).

FCMLT (vectors): Floating-point compare less than vector: an alias of FCM<cc> (vectors).

FCPY: Copy 8-bit floating-point immediate to vector elements (predicated).

FCVT: Floating-point convert precision (predicated).

FCVTZS: Floating-point convert to signed integer, rounding toward zero (predicated).
FCVTZU: Floating-point convert to unsigned integer, rounding toward zero (predicated).
FDIV: Floating-point divide by vector (predicated).
FDIVR: Floating-point reversed divide by vector (predicated).
FDUP: Broadcast 8-bit floating-point immediate to vector elements (unpredicated).
FEXPA: Floating-point exponential accelerator.
FMAD: Floating-point fused multiply-add vectors (predicated), writing multiplicand \(Zdn = Za + Zdn \times Zm\).
FMAX (immediate): Floating-point maximum with immediate (predicated).
FMAX (vectors): Floating-point maximum (predicated).
FMAXNM (immediate): Floating-point maximum number with immediate (predicated).
FMAXNM (vectors): Floating-point maximum number (predicated).
FMAXNVM: Floating-point maximum number recursive reduction to scalar.
FMAXV: Floating-point maximum recursive reduction to scalar.
FMIN (immediate): Floating-point minimum with immediate (predicated).
FMIN (vectors): Floating-point minimum (predicated).
FMINNM (immediate): Floating-point minimum number with immediate (predicated).
FMINNM (vectors): Floating-point minimum number (predicated).
FMINNVM: Floating-point minimum number recursive reduction to scalar.
FMINV: Floating-point minimum recursive reduction to scalar.
FMLA (indexed): Floating-point fused multiply-add by indexed elements \(Zda = Zda + Zn \times Zm\[indexed]\).
FMLA (vectors): Floating-point fused multiply-add vectors (predicated), writing addend \(Zda = Zda + Zn \times Zm\).
FMLS (indexed): Floating-point fused multiply-subtract by indexed elements \(Zda = Zda + -Zn \times Zm\[indexed]\).
FMLS (vectors): Floating-point fused multiply-subtract vectors (predicated), writing addend \(Zda = Zda + -Zn \times Zm\).
FMMLA: Floating-point matrix multiply-accumulate.
FMOV (immediate, predicated): Move 8-bit floating-point immediate to vector elements (predicated): an alias of FCPY.
FMOV (immediate, unpredicated): Move 8-bit floating-point immediate to vector elements (unpredicated): an alias of FDUP.
FMOV (zero, predicated): Move floating-point +0.0 to vector elements (predicated): an alias of CPY (immediate, merging).
FMOV (zero, unpredicated): Move floating-point +0.0 to vector elements (unpredicated): an alias of DUP (immediate).
FMSB: Floating-point fused multiply-subtract vectors (predicated), writing multiplicand \(Zdn = Za + -Zdn \times Zm\).
FMUL (immediate): Floating-point multiply by immediate (predicated).
FMUL (indexed):Floating-point multiply by indexed elements.
FMUL (vectors, predicated): Floating-point multiply vectors (predicated).
FMUL (vectors, unpredicated): Floating-point multiply vectors (unpredicated).
FMULX: Floating-point multiply-extended vectors (predicated).
FNEG: Floating-point negate (predicated).
FNMAD: Floating-point negated fused multiply-add vectors (predicated), writing multiplicand \(Zdn = -Za + -Zdn \times Zm\).
**FNMLA**: Floating-point negated fused multiply-add vectors (predicated), writing addend \([Zda = -Zda + -Zn \times Zm]\).

**FNMLS**: Floating-point negated fused multiply-subtract vectors (predicated), writing addend \([Zda = -Zda + Zn \times Zm]\).

**FNMSB**: Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand \([Zdn = -Za + Zdn \times Zm]\).

**FRECPE**: Floating-point reciprocal estimate (unpredicated).

**FRECPS**: Floating-point reciprocal step (unpredicated).

**FRECPX**: Floating-point reciprocal exponent (predicated).

**FRINT< r >**: Floating-point round to integral value (predicated).

**FRSORTE**: Floating-point reciprocal square root estimate (unpredicated).

**FRSORTS**: Floating-point reciprocal square root step (unpredicated).

**FScale**: Floating-point adjust exponent by vector (predicated).

**FRINT (immediate)**: Floating-point reciprocal estimate (unpredicated).

**FRINT (vectors, predicated)**: Floating-point reciprocal subtract vectors (predicated).

**FRINT (vectors, unpredicated)**: Floating-point reciprocal subtract vectors (unpredicated).

**FSUBR (immediate)**: Floating-point reversed subtract from immediate (predicated).

**FSUBR (vectors)**: Floating-point reversed subtract vectors (predicated).

**FTMAD**: Floating-point trigonometric multiply-add coefficient.

**FTSMUL**: Floating-point trigonometric starting value.

**FTSSEL**: Floating-point trigonometric select coefficient.

**INCB, INCD, INCH, INCW (scalar)**: Increment scalar by multiple of predicate constraint element count.

**INCD, INCH, INCW (vector)**: Increment vector by multiple of predicate constraint element count.

**INCP (scalar)**: Increment scalar by count of true predicate elements.

**INCP (vector)**: Increment vector by count of true predicate elements.

**INDEX (immediate, scalar)**: Create index starting from immediate and incremented by general-purpose register.

**INDEX (immediates)**: Create index starting from and incremented by immediate.

**INDEX (scalar, immediate)**: Create index starting from general-purpose register and incremented by immediate.

**INDEX (scalars)**: Create index starting from and incremented by general-purpose register.

**INSR**: Insert general-purpose register in shifted vector.

**INSR (SIMD&FP scalar)**: Insert SIMD&FP scalar register in shifted vector.

**LASTA (scalar)**: Extract element after last to general-purpose register.

**LASTA (SIMD&FP scalar)**: Extract element after last to SIMD&FP scalar register.

**LASTB (scalar)**: Extract last element to general-purpose register.

**LASTB (SIMD&FP scalar)**: Extract last element to SIMD&FP scalar register.

**LD1B (scalar plus immediate)**: Contiguous load unsigned bytes to vector (immediate index).

**LD1B (scalar plus scalar)**: Contiguous load unsigned bytes to vector (scalar index).

**LD1B (scalar plus vector)**: Gather load unsigned bytes to vector (vector index).
LD1B (vector plus immediate): Gather load unsigned bytes to vector (immediate index).
LD1D (scalar plus immediate): Contiguous load doublewords to vector (immediate index).
LD1D (scalar plus scalar): Contiguous load doublewords to vector (scalar index).
LD1D (vector plus vector): Gather load doublewords to vector (vector index).
LD1D (vector plus immediate): Gather load doublewords to vector (immediate index).
LD1H (scalar plus immediate): Contiguous load unsigned halfwords to vector (immediate index).
LD1H (scalar plus scalar): Contiguous load unsigned halfwords to vector (scalar index).
LD1H (scalar plus vector): Gather load unsigned halfwords to vector (vector index).
LD1H (vector plus immediate): Gather load unsigned halfwords to vector (immediate index).
LD1RB: Load and broadcast unsigned byte to vector.
LD1RD: Load and broadcast doubleword to vector.
LD1RH: Load and broadcast unsigned halfword to vector.
LD1ROB (scalar plus immediate): Contiguous load and replicate thirty-two bytes (immediate index).
LD1ROB (scalar plus scalar): Contiguous load and replicate thirty-two bytes (scalar index).
LD1ROD (scalar plus immediate): Contiguous load and replicate four doublewords (immediate index).
LD1ROD (scalar plus scalar): Contiguous load and replicate four doublewords (scalar index).
LD1ROH (scalar plus immediate): Contiguous load and replicate sixteen halfwords (immediate index).
LD1ROH (scalar plus scalar): Contiguous load and replicate sixteen halfwords (scalar index).
LD1ROW (scalar plus immediate): Contiguous load and replicate eight words (immediate index).
LD1ROW (scalar plus scalar): Contiguous load and replicate eight words (scalar index).
LD1RQB (scalar plus immediate): Contiguous load and replicate sixteen bytes (immediate index).
LD1RQB (scalar plus scalar): Contiguous load and replicate sixteen bytes (scalar index).
LD1RQD (scalar plus immediate): Contiguous load and replicate two doublewords (immediate index).
LD1RQD (scalar plus scalar): Contiguous load and replicate two doublewords (scalar index).
LD1RQH (scalar plus immediate): Contiguous load and replicate eight halfwords (immediate index).
LD1RQH (scalar plus scalar): Contiguous load and replicate eight halfwords (scalar index).
LD1ROW (scalar plus immediate): Contiguous load and replicate four words (immediate index).
LD1ROW (scalar plus scalar): Contiguous load and replicate four words (scalar index).
LD1RSB: Load and broadcast signed byte to vector.
LD1RSH: Load and broadcast signed halfword to vector.
LD1RSW: Load and broadcast signed word to vector.
LD1RW: Load and broadcast unsigned word to vector.
LD1SB (scalar plus immediate): Contiguous load signed bytes to vector (immediate index).
LD1SB (scalar plus scalar): Contiguous load signed bytes to vector (scalar index).
LD1SB (vector plus vector): Gather load signed bytes to vector (vector index).
LD1SB (vector plus immediate): Gather load signed bytes to vector (immediate index).
LD1SH (scalar plus immediate): Contiguous load signed halfwords to vector (immediate index).
LD1SH (scalar plus scalar): Contiguous load signed halfwords to vector (scalar index).
LD1SH (scalar plus vector): Gather load signed halfwords to vector (vector index).
LD1SH (vector plus immediate): Gather load signed halfwords to vector (immediate index).
LD1SW (scalar plus immediate): Contiguous load signed words to vector (immediate index).
LD1SW (scalar plus scalar): Contiguous load signed words to vector (scalar index).
LD1SW (scalar plus vector): Gather load signed words to vector (vector index).
LD1SW (vector plus immediate): Gather load signed words to vector (immediate index).
LD1W (scalar plus immediate): Contiguous load unsigned words to vector (immediate index).
LD1W (scalar plus scalar): Contiguous load unsigned words to vector (scalar index).
LD1W (scalar plus vector): Gather load unsigned words to vector (vector index).
LD1W (vector plus immediate): Gather load unsigned words to vector (immediate index).
LD2B (scalar plus immediate): Contiguous load two-byte structures to two vectors (immediate index).
LD2B (scalar plus scalar): Contiguous load two-byte structures to two vectors (scalar index).
LD2D (scalar plus immediate): Contiguous load two-doubleword structures to two vectors (immediate index).
LD2D (scalar plus scalar): Contiguous load two-doubleword structures to two vectors (scalar index).
LD2H (scalar plus immediate): Contiguous load two-halfword structures to two vectors (immediate index).
LD2H (scalar plus scalar): Contiguous load two-halfword structures to two vectors (scalar index).
LD2W (scalar plus immediate): Contiguous load two-word structures to two vectors (immediate index).
LD2W (scalar plus scalar): Contiguous load two-word structures to two vectors (scalar index).
LD3B (scalar plus immediate): Contiguous load three-byte structures to three vectors (immediate index).
LD3B (scalar plus scalar): Contiguous load three-byte structures to three vectors (scalar index).
LD3D (scalar plus immediate): Contiguous load three-doubleword structures to three vectors (immediate index).
LD3D (scalar plus scalar): Contiguous load three-doubleword structures to three vectors (scalar index).
LD3H (scalar plus immediate): Contiguous load three-halfword structures to three vectors (immediate index).
LD3H (scalar plus scalar): Contiguous load three-halfword structures to three vectors (scalar index).
LD3W (scalar plus immediate): Contiguous load three-word structures to three vectors (immediate index).
LD3W (scalar plus scalar): Contiguous load three-word structures to three vectors (scalar index).
LD4B (scalar plus immediate): Contiguous load four-byte structures to four vectors (immediate index).
LD4B (scalar plus scalar): Contiguous load four-byte structures to four vectors (scalar index).
LD4D (scalar plus immediate): Contiguous load four-doubleword structures to four vectors (immediate index).
LD4D (scalar plus scalar): Contiguous load four-doubleword structures to four vectors (scalar index).
LD4H (scalar plus immediate): Contiguous load four-halfword structures to four vectors (immediate index).
LD4H (scalar plus scalar): Contiguous load four-halfword structures to four vectors (scalar index).
LD4W (scalar plus immediate): Contiguous load four-word structures to four vectors (immediate index).
LD4W (scalar plus scalar): Contiguous load four-word structures to four vectors (scalar index).
LDFF1B (scalar plus scalar): Contiguous load first-fault unsigned bytes to vector (scalar index).
LDFF1B (scalar plus vector): Gather load first-fault unsigned bytes to vector (vector index).
LDFF1B (vector plus immediate): Gather load first-fault unsigned bytes to vector (immediate index).
LDFF1D (scalar plus scalar): Contiguous load first-fault doublewords to vector (scalar index).
LDFF1D (scalar plus vector): Gather load first-fault doublewords to vector (vector index).
LDFF1D (vector plus immediate): Gather load first-fault doublewords to vector (immediate index).
LDFF1H (scalar plus scalar): Contiguous load first-fault unsigned halfwords to vector (scalar index).
LDFF1H (scalar plus vector): Gather load first-fault unsigned halfwords to vector (vector index).
LDFF1H (vector plus immediate): Gather load first-fault unsigned halfwords to vector (immediate index).
LDFF1SB (scalar plus scalar): Contiguous load first-fault signed bytes to vector (scalar index).
LDFF1SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).
LDFF1SB (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).
LDFF1SH (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).
LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).
LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).
LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).
LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).
LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).
LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).
LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).
LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).
LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).
LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).
LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).
LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).
LDNF1SH: Contiguous load non-fault signed halfwords to vector (immediate index).
LDNF1SW: Contiguous load non-fault signed words to vector (immediate index).
LDNF1W: Contiguous load non-fault unsigned words to vector (immediate index).
LDNT1B (scalar plus immediate): Contiguous load non-temporal bytes to vector (immediate index).
LDNT1B (scalar plus scalar): Contiguous load non-temporal bytes to vector (scalar index).
LDNT1D (scalar plus immediate): Contiguous load non-temporal doublewords to vector (immediate index).
LDNT1D (scalar plus scalar): Contiguous load non-temporal doublewords to vector (scalar index).
LDNT1H (scalar plus immediate): Contiguous load non-temporal halfwords to vector (immediate index).
LDNT1H (scalar plus scalar): Contiguous load non-temporal halfwords to vector (scalar index).
LDNT1W (scalar plus immediate): Contiguous load non-temporal words to vector (immediate index).
LDNT1W (scalar plus scalar): Contiguous load non-temporal words to vector (scalar index).
LDR (predicate): Load predicate register.
LDR (vector): Load vector register.
LSL (immediate, predicated): Logical shift left by immediate (predicated).
LSL (immediate, unpredicated): Logical shift left by immediate (unpredicated).
LSL (vectors): Logical shift left by vector (predicated).
LSL (wide elements, predicated): Logical shift left by 64-bit wide elements (predicated).
LSL (wide elements, unpredicated): Logical shift left by 64-bit wide elements (unpredicated).
LSLR: Reversed logical shift left by vector (predicated).
LSR (immediate, predicated): Logical shift right by immediate (predicated).
LSR (immediate, unpredicated): Logical shift right by immediate (unpredicated).
LSR (vectors): Logical shift right by vector (predicated).
LSR (wide elements, predicated): Logical shift right by 64-bit wide elements (predicated).
LSR (wide elements, unpredicated): Logical shift right by 64-bit wide elements (unpredicated).
LSRR: Reversed logical shift right by vector (predicated).
MAD: Multiply-add vectors (predicated), writing multiplicand \[Zdn = Zn + Zdn * Zm\].
MLA: Multiply-add vectors (predicated), writing addend \[Zda = Zda + Zn * Zm\].
MLS: Multiply-subtract vectors (predicated), writing addend \[Zda = Zda - Zn * Zm\].
MOV: Move logical bitmask immediate to vector elements (unpredicated): an alias of DUPM.
MOV: Move predicate (unpredicated): an alias of ORR (predicates).
MOV (immediate, predicated, merging): Move signed integer immediate to vector elements (merging): an alias of CPY (immediate, merging).
MOV (immediate, predicated, zeroing): Move signed integer immediate to vector elements (zeroing): an alias of CPY (immediate, zeroing).
MOV (immediate, unpredicated): Move signed immediate to vector elements (unpredicated): an alias of DUP (immediate).
MOV (scalar, predicated): Move general-purpose register to vector elements (predicated): an alias of CPY (scalar).
MOV (scalar, unpredicated): Move general-purpose register to vector elements (unpredicated): an alias of DUP (scalar).
MOV (SIMD&FP scalar, unpredicated): Move indexed element or SIMD&FP scalar to vector (unpredicated): an alias of DUP (indexed).
MOVPRFX (predicated): Move prefix (predicated).
MOVPRFX (unpredicated): Move prefix (unpredicated).
MOVS (predicated): Move predicates (zeroing), setting the condition flags: an alias of ANDS.
A64 -- SVE Instructions (alphabetic order)

**MOVS (unpredicated):** Move predicate (unpredicated), setting the condition flags: an alias of ORRS.

**MSB:** Multiply-subtract vectors (predicated), writing multiplicand \([Zdn = Za - Zdn \times Zm]\).

**MUL (immediate):** Multiply by immediate (unpredicated).

**MUL (vectors):** Multiply vectors (predicated).

**NAND:** Bitwise NAND predicates.

**NANDS:** Bitwise NAND predicates, setting the condition flags.

**NEG:** Negate (predicated).

**NOR:** Bitwise NOR predicates.

**NORS:** Bitwise NOR predicates, setting the condition flags.

**NOT (predicate):** Bitwise invert predicate: an alias of EOR (predicates).

**NOT (vector):** Bitwise invert vector (predicated).

**NOTS:** Bitwise invert predicate, setting the condition flags: an alias of EORS.

**ORN (immediate):** Bitwise inclusive OR with inverted immediate (unpredicated): an alias of ORR (immediate).

**ORN (predicates):** Bitwise inclusive OR inverted predicate.

**ORNS:** Bitwise inclusive OR inverted predicate, setting the condition flags.

**ORR (immediate):** Bitwise inclusive OR with immediate (unpredicated).

**ORR (predicates):** Bitwise inclusive OR predicates.

**ORR (vectors, predicated):** Bitwise inclusive OR vectors (predicated).

**ORR (vectors, unpredicated):** Bitwise inclusive OR vectors (unpredicated).

**ORRS:** Bitwise inclusive OR predicates, setting the condition flags.

**ORV:** Bitwise inclusive OR reduction to scalar.

**PFALSE:** Set all predicate elements to false.

**PFIRST:** Set the first active predicate element to true.

**PNEXT:** Find next active predicate.

**PRFB (scalar plus immediate):** Contiguous prefetch bytes (immediate index).

**PRFB (scalar plus scalar):** Contiguous prefetch bytes (scalar index).

**PRFB (scalar plus vector):** Gather prefetch bytes (scalar plus vector).

**PRFB (vector plus immediate):** Gather prefetch bytes (vector plus immediate).

**PRFD (scalar plus immediate):** Contiguous prefetch doublewords (immediate index).

**PRFD (scalar plus scalar):** Contiguous prefetch doublewords (scalar index).

**PRFD (scalar plus vector):** Gather prefetch doublewords (scalar plus vector).

**PRFD (vector plus immediate):** Gather prefetch doublewords (vector plus immediate).

**PRFH (scalar plus immediate):** Contiguous prefetch halfwords (immediate index).

**PRFH (scalar plus scalar):** Contiguous prefetch halfwords (scalar index).

**PRFH (scalar plus vector):** Gather prefetch halfwords (scalar plus vector).

**PRFH (vector plus immediate):** Gather prefetch halfwords (vector plus immediate).
PRFW (scalar plus immediate): Contiguous prefetch words (immediate index).
PRFW (scalar plus scalar): Contiguous prefetch words (scalar index).
PRFW (scalar plus vector): Gather prefetch words (scalar plus vector).
PRFW (vector plus immediate): Gather prefetch words (vector plus immediate).
PTEST: Set condition flags for predicate.
PTRUE: Initialise predicate from named constraint.
PTRUES: Initialise predicate from named constraint and set the condition flags.
PUNPKHI, PUNPKLO: Unpack and widen half of predicate.
RBIT: Reverse bits (predicated).
RDFFR (predicated): Return predicate of successfully loaded elements.
RDFFR (unpredicated): Read the first-fault register.
RDFFRS: Return predicate of successfully loaded elements, setting the condition flags.
RDVL: Read multiple of vector register size to scalar register.
REV (predicate): Reverse all elements in a predicate.
REV (vector): Reverse all elements in a vector (unpredicated).
REVb, REVh, REVw: Reverse bytes / halfwords / words within elements (predicated).
SABD: Signed absolute difference (predicated).
SADDV: Signed add reduction to scalar.
SCVTF: Signed integer convert to floating-point (predicated).
SDIV: Signed divide (predicated).
SDIVR: Signed reversed divide (predicated).
SDOT (indexed): Signed integer indexed dot product.
SDOT (vectors): Signed integer dot product.
SEL (predicates): Conditionally select elements from two predicates.
SEL (vectors): Conditionally select elements from two vectors.
SETFFR: Initialise the first-fault register to all true.
SMAX (immediate): Signed maximum with immediate (unpredicated).
SMAX (vectors): Signed maximum vectors (predicated).
SMAXV: Signed maximum reduction to scalar.
SMIN (immediate): Signed minimum with immediate (unpredicated).
SMIN (vectors): Signed minimum vectors (predicated).
SMINY: Signed minimum reduction to scalar.
SMMLA: Signed integer matrix multiply-accumulate.
SMULH: Signed multiply returning high half (predicated).
SPlice: Splice two vectors under predicate control.
SQADD (immediate): Signed saturating add immediate (unpredicated).
**SQADD (vectors)**: Signed saturating add vectors (unpredicated).

**SQDECB**: Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.

**SQDECD (scalar)**: Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

**SQDECD (vector)**: Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

**SQDECH (scalar)**: Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.

**SQDECH (vector)**: Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.

**SQDECP (scalar)**: Signed saturating decrement scalar by count of true predicate elements.

**SQDECP (vector)**: Signed saturating decrement vector by count of true predicate elements.

**SQDECW (scalar)**: Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

**SQDECW (vector)**: Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

**SQINCB**: Signed saturating increment scalar by multiple of 8-bit predicate constraint element count.

**SQINCD (scalar)**: Signed saturating increment scalar by multiple of 64-bit predicate constraint element count.

**SQINCD (vector)**: Signed saturating increment vector by multiple of 64-bit predicate constraint element count.

**SQINCH (scalar)**: Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

**SQINCH (vector)**: Signed saturating increment vector by multiple of 16-bit predicate constraint element count.

**SQINCP (scalar)**: Signed saturating increment scalar by count of true predicate elements.

**SQINCP (vector)**: Signed saturating increment vector by count of true predicate elements.

**SQINCW (scalar)**: Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

**SQINCW (vector)**: Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

**SQSUB (immediate)**: Signed saturating subtract immediate (unpredicated).

**SQSUB (vectors)**: Signed saturating subtract vectors (unpredicated).

**ST1B (scalar plus immediate)**: Contiguous store bytes from vector (immediate index).

**ST1B (scalar plus scalar)**: Contiguous store bytes from vector (scalar index).

**ST1B (scalar plus vector)**: Scatter store bytes from a vector (vector index).

**ST1B (vector plus immediate)**: Scatter store bytes from a vector (immediate index).

**ST1D (scalar plus immediate)**: Contiguous store doublewords from vector (immediate index).

**ST1D (scalar plus scalar)**: Contiguous store doublewords from vector (scalar index).

**ST1D (scalar plus vector)**: Scatter store doublewords from a vector (vector index).

**ST1D (vector plus immediate)**: Scatter store doublewords from a vector (immediate index).

**ST1H (scalar plus immediate)**: Contiguous store halfwords from vector (immediate index).

**ST1H (scalar plus scalar)**: Contiguous store halfwords from vector (scalar index).

**ST1H (scalar plus vector)**: Scatter store halfwords from a vector (vector index).

**ST1H (vector plus immediate)**: Scatter store halfwords from a vector (immediate index).

**ST1W (scalar plus immediate)**: Contiguous store words from vector (immediate index).

**ST1W (scalar plus scalar)**: Contiguous store words from vector (scalar index).

**ST1W (scalar plus vector)**: Scatter store words from a vector (vector index).
ST1W (vector plus immediate): Scatter store words from a vector (immediate index).

ST2B (scalar plus immediate): Contiguous store two-byte structures from two vectors (immediate index).

ST2B (scalar plus scalar): Contiguous store two-byte structures from two vectors (scalar index).

ST2D (scalar plus immediate): Contiguous store two-doubleword structures from two vectors (immediate index).

ST2D (scalar plus scalar): Contiguous store two-doubleword structures from two vectors (scalar index).

ST2H (scalar plus immediate): Contiguous store two-halfword structures from two vectors (immediate index).

ST2H (scalar plus scalar): Contiguous store two-halfword structures from two vectors (scalar index).

ST2W (scalar plus immediate): Contiguous store two-word structures from two vectors (immediate index).

ST2W (scalar plus scalar): Contiguous store two-word structures from two vectors (scalar index).

ST3B (scalar plus immediate): Contiguous store three-byte structures from three vectors (immediate index).

ST3B (scalar plus scalar): Contiguous store three-byte structures from three vectors (scalar index).

ST3D (scalar plus immediate): Contiguous store three-doubleword structures from three vectors (immediate index).

ST3D (scalar plus scalar): Contiguous store three-doubleword structures from three vectors (scalar index).

ST3H (scalar plus immediate): Contiguous store three-halfword structures from three vectors (immediate index).

ST3H (scalar plus scalar): Contiguous store three-halfword structures from three vectors (scalar index).

ST3W (scalar plus immediate): Contiguous store three-word structures from three vectors (immediate index).

ST3W (scalar plus scalar): Contiguous store three-word structures from three vectors (scalar index).

ST4B (scalar plus immediate): Contiguous store four-byte structures from four vectors (immediate index).

ST4B (scalar plus scalar): Contiguous store four-byte structures from four vectors (scalar index).

ST4D (scalar plus immediate): Contiguous store four-doubleword structures from four vectors (immediate index).

ST4D (scalar plus scalar): Contiguous store four-doubleword structures from four vectors (scalar index).

ST4H (scalar plus immediate): Contiguous store four-halfword structures from four vectors (immediate index).

ST4H (scalar plus scalar): Contiguous store four-halfword structures from four vectors (scalar index).

ST4W (scalar plus immediate): Contiguous store four-word structures from four vectors (immediate index).

ST4W (scalar plus scalar): Contiguous store four-word structures from four vectors (scalar index).

STNT1B (scalar plus immediate): Contiguous store non-temporal bytes from vector (immediate index).

STNT1B (scalar plus scalar): Contiguous store non-temporal bytes from vector (scalar index).

STNT1D (scalar plus immediate): Contiguous store non-temporal doublewords from vector (immediate index).

STNT1D (scalar plus scalar): Contiguous store non-temporal doublewords from vector (scalar index).

STNT1H (scalar plus immediate): Contiguous store non-temporal halfwords from vector (immediate index).

STNT1H (scalar plus scalar): Contiguous store non-temporal halfwords from vector (scalar index).

STNT1W (scalar plus immediate): Contiguous store non-temporal words from vector (immediate index).

STNT1W (scalar plus scalar): Contiguous store non-temporal words from vector (scalar index).

STR (predicate): Store predicate register.

STR (vector): Store vector register.

SUB (immediate): Subtract immediate (unpredicated).
SUB (vectors, predicated): Subtract vectors (predicated).
SUB (vectors, unpredicated): Subtract vectors (unpredicated).
SUBR (immediate): Reversed subtract from immediate (unpredicated).
SUBR (vectors): Reversed subtract vectors (predicated).

SUDOT: Signed by unsigned integer indexed dot product.
SUNPKHI, SUNPKLO: Signed unpack and extend half of vector.
SXTB, SXTH, SXTW: Signed byte / halfword / word extend (predicated).
TBL: Programmable table lookup in single vector table.
TRN1, TRN2 (predicates): Interleave even or odd elements from two predicates.
TRN1, TRN2 (vectors): Interleave even or odd elements from two vectors.

UABD: Unsigned absolute difference (predicated).
UADDV: Unsigned add reduction to scalar.
UCVTF: Unsigned integer convert to floating-point (predicated).
UDIV: Unsigned divide (predicated).
UDIVR: Unsigned reversed divide (predicated).
UDOT (indexed): Unsigned integer indexed dot product.
UDOT (vectors): Unsigned integer dot product.
UMAX (immediate): Unsigned maximum with immediate (unpredicated).
UMAX (vectors): Unsigned maximum vectors (predicated).
UMAXV: Unsigned maximum reduction to scalar.
UMAXV: Unsigned maximum reduction to scalar.
UMIN (immediate): Unsigned minimum with immediate (unpredicated).
UMIN (vectors): Unsigned minimum vectors (predicated).
UMINV: Unsigned minimum reduction to scalar.
UMMLA: Unsigned integer matrix multiply-accumulate.
UMULH: Unsigned multiply returning high half (predicated).
UQADD (immediate): Unsigned saturating add immediate (unpredicated).
UQADD (vectors): Unsigned saturating add vectors (unpredicated).
UQDECB: Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.
UQDECD (scalar): Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.
UQDECD (vector): Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.
UQDECH (scalar): Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.
UQDECH (vector): Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count.
UQDECP (scalar): Unsigned saturating decrement scalar by count of true predicate elements.
UQDECP (vector): Unsigned saturating decrement vector by count of true predicate elements.
UQDECW (scalar): Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.
UQDECW (vector): Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count.
UQINCB: Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count.

UQINCD (scalar): Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

UQINCD (vector): Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

UQINCH (scalar): Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

UQINCH (vector): Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

UQINCP (scalar): Unsigned saturating increment scalar by count of true predicate elements.

UQINCP (vector): Unsigned saturating increment vector by count of true predicate elements.

UQINCW (scalar): Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count.

UQINCW (vector): Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count.

UQSUB (immediate): Unsigned saturating subtract immediate (unpredicated).

UQSUB (vectors): Unsigned saturating subtract vectors (unpredicated).

USDOT (indexed): Unsigned by signed integer indexed dot product.

USDOT (vectors): Unsigned by signed integer dot product.

USMMLA: Unsigned by signed integer matrix multiply-accumulate.

UUNPKHI, UUNPKLO: Unsigned unpack and extend half of vector.

UXTB, UXTH, UXTW: Unsigned byte / halfword / word extend (predicated).

UZP1, UZP2 (predicates): Concatenate even or odd elements from two predicates.

UZP1, UZP2 (vectors): Concatenate even or odd elements from two vectors.

WHILELE: While incrementing signed scalar less than or equal to scalar.

WHILELO: While incrementing unsigned scalar lower than scalar.

WHILELS: While incrementing unsigned scalar lower or same as scalar.

WHILELT: While incrementing signed scalar less than scalar.

WRFFR: Write the first-fault register.

ZIP1, ZIP2 (predicates): Interleave elements from two half predicates.

ZIP1, ZIP2 (vectors): Interleave elements from two half vectors.
ABS

Absolute value (predicated)

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 0 | size  0 1 0 | 1 1 0 1 0 1 | Pg  Zn  Zd |
```

ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in "size":
  - | size | <T> |
    | 00   | B   |
    | 01   | H   |
    | 10   | S   |
    | 11   | D   |
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand, e, esize]);
    element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADD (immediate)

Add immediate (unpredicated)

Add an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is ".#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as ".#0, LSL #8".

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 0 1 0 0 1 0 1 | size | 1 0 0 | 0 0 0 1 1 | sh | imm8 | Zdn |

ADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the “imm8” field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 + imm;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADD (vectors, predicated)

Add vectors (predicated)

Add active elements of the second source vector to corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| size | Pg | Zm | Zdn |

ADD <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 + element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADD (vectors, unpredicated)

Add vectors (unpredicated)

Add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
ADD <Zd>, <Zn>, <Zm>
```

If !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = element1 + element2;
Z[d] = result;
```
ADDPL

Add multiple of predicate register size to scalar register

Add the current predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | Rn | 0  | 1  | 0  | 1  | 0  | imm6 | Rd |

ADDPL <Xd|SP>, <Xn|SP>, #<imm>

if !HaveSVE() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) result = operand1 + (imm * (PL DIV 8));
if d == 31 then
    SP[] = result;
else
    X[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADDVL

Add multiple of vector register size to scalar register

Add the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  0  1  0  0  0  0  1  Rn  0  1  0  1  0  imm6  Rd
```

ADDVL <Xd|SP>, <Xn|SP>, #<imm>

if !HaveSVE() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler Symbols

<Xd|SP>    Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>    Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>    Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) result = operand1 + (imm * (VL DIV 8));
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADR

Compute vector address

Optionally sign or zero-extend the least significant 32-bits of each element from a vector of offsets or indices in the
second source vector, scale each index by 2, 4 or 8, add to a vector of base addresses from the first source vector, and
place the resulting addresses in the destination vector. This instruction is unpredicated.

It has encodings from 3 classes: Packed offsets, Unpacked 32-bit signed offsets and Unpacked 32-bit unsigned offsets

Packed offsets

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | sz | 1  | Zm | 1  | 0  | 1  | 0  | msz | Zn | Zd |

ADR <Zd>.<T>, [<Zn>.<T>, <Zm>.<T>{, <mod> <amount>}]

if !HaveSVE() then UNDEFINED;
  integer esize = 32 << UInt(sz);
  integer n = UInt(Zn);
  integer m = UInt(Zm);
  integer d = UInt(Zd);
  integer osize = esize;
  boolean unsigned = TRUE;
  integer mbytes = 1 << UInt(msz);

Unpacked 32-bit signed offsets

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | Zm | 1  | 0  | 1  | 0  | msz | Zn | Zd |

ADR <Zd>.D, [<Zn>.D, <Zm>.D, SXTW{ <amount>}] 

if !HaveSVE() then UNDEFINED;
  integer esize = 64;
  integer n = UInt(Zn);
  integer m = UInt(Zm);
  integer d = UInt(Zd);
  integer osize = 32;
  boolean unsigned = FALSE;
  integer mbytes = 1 << UInt(msz);

Unpacked 32-bit unsigned offsets

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | Zm | 1  | 0  | 1  | 0  | msz | Zn | Zd |

ADR <Zd>.D, [<Zn>.D, <Zm>.D, UXTW{ <amount>}] 

if !HaveSVE() then UNDEFINED;
  integer esize = 64;
  integer n = UInt(Zn);
  integer m = UInt(Zm);
  integer d = UInt(Zd);
  integer osize = 32;
  boolean unsigned = TRUE;
  integer mbytes = 1 << UInt(msz);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the base scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod> Is the index extend and shift specifier, encoded in “msz”:

<table>
<thead>
<tr>
<th>msz</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>[absent]</td>
</tr>
<tr>
<td>x1</td>
<td>LSL</td>
</tr>
<tr>
<td>10</td>
<td>LSL</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, encoded in “msz”:

<table>
<thead>
<tr>
<th>msz</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>[absent]</td>
</tr>
<tr>
<td>01</td>
<td>#1</td>
</tr>
<tr>
<td>10</td>
<td>#2</td>
</tr>
<tr>
<td>11</td>
<td>#3</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) offs = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) addr = Elem[base, e, esize];
    integer offset = Int(Elem[offs, e, esize]<osize-1:0>, unsigned);
    Elem[result, e, esize] = addr + (offset * mbytes);
Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AND (immediate)

Bitwise AND with immediate (unpredicated)

Bitwise AND an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction BIC (immediate).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 1 0 0 0 0 imm13</td>
</tr>
<tr>
<td>Zdn</td>
</tr>
</tbody>
</table>

AND <Zdn>.<T>, <Zdn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>:”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxxxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xxxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 64;
bits(VL) operand = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(64) element1 = Elem[operand, e, 64];
  Elem[result, e, 64] = element1 AND imm;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AND (predicates)

Bitwise AND predicates

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias MOV (predicate, predicated, zeroing).


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler Symbols

< Pd > Is the name of the destination scalable predicate register, encoded in the "Pd" field.
< Pg > Is the name of the governing scalable predicate register, encoded in the "Pg" field.
< Pn > Is the name of the first source scalable predicate register, encoded in the "Pn" field.
< Pm > Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicate, predicated, zeroing)</td>
<td>S == '0' &amp;&amp; Pn == Pm</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
**AND (vectors, predicated)**

Bitwise AND vectors (predicated)

Bitwise AND active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 0 1 0 0 | size | 0 1 1 | 0 1 | 0 0 0 0 | Pg  | Zm  | Zdn
```

AND `<Zdn>.<T>`, `<Pg>/M`, `<Zdn>.<T>`, `<Zm>.<T>`

if `HaveSVE()` then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 AND element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
AND (vectors, unpredicated)

Bitwise AND vectors (unpredicated)

Bitwise AND all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 0 0 1</td>
</tr>
</tbody>
</table>

**AND <Zd>.D, <Zn>.D, <Zm>.D**

```plaintext
if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

- **<Zd>** is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<Zn>** is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
Z[d] = operand1 AND operand2;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ANDS

Bitwise AND predicates, setting the condition flags

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias **MOV$S$ (predicated)**.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1
S

```

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

**Assembler Symbols**

<**Pd**> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<**Pg**> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<**Pn**> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<**Pm**> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVS (predicated)</td>
<td>S == '1' &amp;&amp; Pn == Pm</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```
ANDV

**Bitwise AND reduction to scalar**

Bitwise AND horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as all ones.

![Vector and scalar register layout]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |

**Operation**

```assembly
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt (size);
integer g = UInt (Pg);
integer n = UInt (Zn);
integer d = UInt (Vd);

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

```
ASR (immediate, predicated)

Arithmetic shift right by immediate (predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 1 0 0 | tszh 0 0 0 0 0 0 0 | Pg 0 1 0 0 | Zdn
L U
```

ASR <Zdn>.<T>, <Pg>/M, <Zdn>..<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

```
tszh tspzl <T>
00 00 RESERVED
00 01 B
00 1x H
01 xx S
1x xx D
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = ASR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ASR (immediate, unpredicated)

Arithmetic shift right by immediate (unpredicated)

Shift right by immediate, preserving the sign bit, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

ASR <Zd>.<T>, <Zn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = ASR(element1, shift);

Z[d] = result;

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ASR (vectors)

Arithmetic shift right by vector (predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

ASR (vectors)
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ASR (wide elements, predicated)

Arithmetic shift right by 64-bit wide elements (predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

```asm
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

- `CheckSVEEnabled();`
- `integer elements = VL DIV esize;`
- `bits(PL) mask = P[g];`
- `bits(VL) operand1 = Z[dn];`
- `bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();`
- `bits(VL) result;`
- `for e = 0 to elements-1`
  - `if Elem[mask, e, esize] == '1' then`
    - `bits(esize) element1 = Elem[operand1, e, esize];`
    - `bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];`
    - `integer shift = Min(UInt(element2), esize);`
    - `Elem[result, e, esize] = ASR(element1, shift);`
  - `else`
    - `Elem[result, e, esize] = Elem[operand1, e, esize];`
- `Z[dn] = result;`

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ASR (wide elements, unpredicated)

Arithmetic shift right by 64-bit wide elements (unpredicated)

Shift right, preserving the sign bit, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & Zm & 1 & 0 & 0 & 0 & 0 & 0 & Zn & Zd
\end{array}
\]

ASR \(<Zd>\), \(<Zn>.\), \(<Zm>.\).

if !\(\text{HaveSVE}()\) then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << \(\text{UInt}(\text{size})\);
integer n = \(\text{UInt}(Zn)\);
integer m = \(\text{UInt}(Zm)\);
integer d = \(\text{UInt}(Zd)\);

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\(\text{VL}\)) operand1 = \(Z[n]\);
bits(\(\text{VL}\)) operand2 = \(Z[m]\);
bits(\(\text{VL}\)) result;

for e = 0 to elements-1
    bits(esize) element1 = \(\text{Elem}[\text{operand1}, e, \text{esize}]\);
    bits(64) element2 = \(\text{Elem}[\text{operand2}, (e * \text{esize}) \div \text{64}, 64]\);
    integer shift = \(\text{Min}([\text{UInt}(\text{element2}), \text{esize}]\));
    \(\text{Elem}[\text{result}, e, \text{esize}] = \text{ASR}(\text{element1}, \text{shift})\);

\(Z[d] = \text{result}\);
ASRD

Arithmetic shift right for divide by immediate (predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The result rounds toward zero as in a signed division. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |

ASRD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```plaintext
if !HaveSVE() then UNDEFINED;
bots(4) tsize = tzh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tzh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tzh:imm3”.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bots(PL) mask = P[g];
bots(VL) operand1 = Z[dn];
bots(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element1 = SInt(Elem[operand1, e, esize]);
    if element1 < 0 then
      element1 = element1 + ((1 << shift) - 1);
      Elem[result, e, esize] = (element1 >> shift)<esize-1:0>;
    else
      Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    Z[dn] = result;
```

Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Reversed arithmetic shift right by vector (predicated)

Reversed shift right, preserving the sign bit, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = ASR(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point down convert to BFloat16 format (predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the result type is smaller than the input type, the results are zero-extended to fill each destination element.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE (FEAT_BF16)

| 0 1 1 0 0 1 0 1 1 0 0 1 0 1 0 1 0 1 83 | H | 77 0 1 | M | 127 | S |

if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, 32] == '1' then
    bits(32) element = Elem[operand, e, 32];
    Elem[result, 2*e, 16] = FPConvertBF(element, FPCR[]);
    Elem[result, 2*e+1, 16] = Zeros();

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFCVTNT

Floating-point down convert and narrow to BFloat16 (top, predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

BFCVTNT <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, 32] == '1' then
    bits(32) element = Elem[operand, e, 32];
    Elem[result, 2*e+1, 16] = FPConvertBF(element, FPCR[]);
  Z[d] = result;
BFDOT (indexed)

BFloat16 floating-point indexed dot product

Irrespective of the control bits in the FPCR, this instruction:
* Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
* Uses the non-IEEE Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
* Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
* Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
* Only the Default NaN is generated, as if FPCR.DN had the value 1.

The BFloat16 pairs within the second source vector are specified using an immediate index which selects the same BFloat16 pair position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE (FEAT_BF16)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 0 1 1 i2 Zm 0 1 0 0 0 0 Zn Zda</td>
</tr>
</tbody>
</table>


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
<imm> Is the immediate index, in the range 0 to 3, encoded in the “i2” field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} 32;
integer eltspersegment = 128 \texttt{DIV} 32;
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) operand3 = \texttt{Z}[da];
bits(\texttt{VL}) result;

for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(16) elt1_a = \texttt{Elem}[operand1, 2 \times e + 0, 16];
  bits(16) elt1_b = \texttt{Elem}[operand1, 2 \times e + 1, 16];
  bits(16) elt2_a = \texttt{Elem}[operand2, 2 \times s + 0, 16];
  bits(16) elt2_b = \texttt{Elem}[operand2, 2 \times s + 1, 16];
  bits(32) sum = \texttt{Elem}[operand3, e, 32];
  \texttt{BFDotAdd}(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
  \texttt{Elem}[result, e, 32] = sum;
\texttt{Z}[da] = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \texttt{MOVPRFX} instruction must be unpredicated.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**BFDOT (vectors)**

BFloat16 floating-point dot product

Irrespective of the control bits in the FPCR, this instruction:
* Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
* Uses the non-IEEE Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
* Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
* Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
* Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
* Only the Default NaN is generated, as if FPCR.DN had the value 1.
This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

**SVE (FEAT_BF16)**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 0 0 1 0 0 0 1 1 Zm | 1 0 0 0 0 0 Zn | Zda
```

**BFDOT <Zda>.S, <Zn>.H, <Zm>.H**

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler Symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;
Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFMLALB (indexed)

BFLOAT16 floating-point multiply-add long to single-precision (bottom, indexed)

This BFLOAT16 floating-point multiply-add long instruction widens the even-numbered BFLOAT16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
( FEAT_BF16 )

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | i3h | Zm | 0  | 1  | 0  | 0  | i3l | Zn | Zda | 0  | 1  | 0  | 0  | i3h | Zm | 0  | 1  | 0  | 0  | i3l | 0  |


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm> Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(32) element1 = Elem[operand1, 2 * e + 0, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, s, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLALB (vectors)

BFLOAT16 floating-point multiply-add long to single-precision (bottom)

This BFLOAT16 floating-point multiply-add long instruction widens the even-numbered BFLOAT16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the source vectors. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(Feat_BF16)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | Zm | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Zn | 1 | 0 | 0 | 0 | 0 | 0 | Zda |


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt (Zn);
integer m = UInt (Zm);
integer da = UInt (Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + 0, 16] : Zeros (16);
    bits(32) element2 = Elem[operand2, 2 * e + 0, 16] : Zeros (16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[ ]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLALT (indexed)

BFLOAT16 floating-point multiply-add long to single-precision (top, indexed)

This BFLOAT16 floating-point multiply-add long instruction widens the odd-numbered BFLOAT16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)

\[
\begin{array}{cccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & i3h & Zm & 0 & 1 & 0 & 0 & i3l & 1 & Zn & Zda & 0 & 2 & op & T
\end{array}
\]


\[
\begin{array}{l}
\text{if } \text{!HaveSVE()} || \text{!HaveBF16Ext()} \text{ then UNDEFINED;} \\
\text{integer } n = \text{UInt}(Zn); \\
\text{integer } m = \text{UInt}(Zm); \\
\text{integer } da = \text{UInt}(Zda); \\
\text{integer } index = \text{UInt}(i3h:i3l);
\end{array}
\]

Assembler Symbols

\texttt{
<Zda>} Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

\texttt{
<Zn>} Is the name of the first source scalable vector register, encoded in the "Zn" field.

\texttt{
<Zm>} Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

\texttt{<imm>} Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

\[
\text{CheckSVEEnabled();} \\
\text{integer } \text{elements} = \text{VL} \text{ DIV 32}; \\
\text{integer } \text{eltspersegment} = 128 \text{ DIV 32}; \\
\text{bits(VL)} \text{ operand1} = Z[n]; \\
\text{bits(VL)} \text{ operand2} = Z[m]; \\
\text{bits(VL)} \text{ operand3} = Z[da]; \\
\text{bits(VL)} \text{ result}; \\
\text{for } e = 0 \text{ to elements-1} \\
\text{integer } \text{segmentbase} = e - (e \text{ MOD eltspersegment}); \\
\text{integer } s = 2 \times \text{segmentbase} + \text{index}; \\
\text{bits(32)} \text{ element1} = \text{Elem}[\text{operand1, 2} \times e + 1, 16] : Zeros(16); \\
\text{bits(32)} \text{ element2} = \text{Elem}[\text{operand2, s, 16}] : Zeros(16); \\
\text{bits(32)} \text{ element3} = \text{Elem}[\text{operand3, e, 32}]; \\
\text{Elem}[\text{result, e, 32}] = \text{BFMatrixAdd(element3, element1, element2, FPCR[])}; \\
Z[da] = \text{result};
\]

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLALT (vectors)

BFLOAT16 floating-point multiply-add long to single-precision (top)

This BFLOAT16 floating-point multiply-add long instruction widens the odd-numbered BFLOAT16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFLOAT16 elements in the source vectors. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zn</td>
<td>1</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

o2 op T


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + 1, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + 1, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BF MulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMMMLA

BFLOAT16 floating-point matrix multiply-accumulate

Irrespective of the control bits in the FPCR, this instruction:
* Performs two unfused sums-of-products within each two pairs of adjacent BFLOAT16 elements while multiplying the 2×4 matrix of BFLOAT16 values held in each 128-bit segment of the first source vector by the 4×2 matrix of BFLOAT16 values in the corresponding segment of the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2×2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
* Uses the non-IEEE Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
* Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
* Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
* Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
* Only the Default NaN is generated, as if FPCR.DN had the value 1.

This instruction is unpredicated and vector length agnostic.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

**SVE**

(HECK_BF16)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | Zm | 1  | 1  | 1  | 0  | 0  | 1  | Zn | 1  | Zda |


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CHECKSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1
  op1 = Elem[operand1, s, 128];
  op2 = Elem[operand2, s, 128];
  addend = Elem[operand3, s, 128];
  res = BFMatMulAdd(addend, op1, op2);  
  Elem[result, s, 128] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**BIC (immediate)**

Bitwise clear bits using immediate (unpredicated)

Bitwise clear bits using immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of **AND (immediate)**. This means:

- The encodings in this description are named to match the encodings of **AND (immediate)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **AND (immediate)** gives the operational pseudocode for this instruction.

```
0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 | imm13   |
```

**BIC <Zdn>.<T>, <Zdn>.<T>, #<const>**  
is equivalent to  
**AND <Zdn>.<T>, <Zdn>.<T>, #(<const> - 1)**

**Assembler Symbols**

- `<Zdn>`: Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<const>`: Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

The description of **AND (immediate)** gives the operational pseudocode for this instruction.

**Operational information**

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**BIC (predicates)**

Bitwise clear predicates

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

```
```

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 AND (NOT element2);
  else
    ElemP[result, e, esize] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```
**BIC (vectors, predicated)**

Bitwise AND inverted active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | Pg | Zm | Zdn |

**BIC** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
it
egar elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 AND (NOT element2);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
  Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BIC (vectors, unpredicated)

Bitwise clear vectors (unpredicated)

Bitwise AND inverted all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & Zm & 0 & 0 & 1 & 1 & 0 & 0 & Zn & Zd
\end{array}
\]

BIC <Zd>.D, <Zn>.D, <Zm>.D

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];

Z[d] = operand1 AND (NOT operand2);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BICS

Bitwise clear predicates, setting the condition flags

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
icnteger elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
      ElemP[result, e, esize] = element1 AND (NOT element2);
    else
      ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**BRKA**

Break after first true condition

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**BRKA** `<Pd>.B`, `<Pg>/<ZM>`, `<Pn>.B`

if `!HaveSVE()` then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

**Assembler Symbols**

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<ZM>` is the predication qualifier, encoded in "M":

<table>
<thead>
<tr>
<th></th>
<th><code>&lt;ZM&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>
- `<Pn>` is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if element
        ElemP[result, e, esize] = if !break then '1' else '0';
        break = break || element;
    elseif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';
    endif
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Break after first true condition, setting the condition flags

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 Pg 0 Pn 0 Pd</td>
</tr>
</tbody>
</table>

**BRKAS**<Pd>.B, <Pg>/Z, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

**Assembler Symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

CheckSVEEnabled();
integer elements = VLAN esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
  boolean element = ElemP[operand, e, esize] == '1';
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = if !break then '1' else '0';
    break = break || element;
  elsif merging then
    ElemP[result, e, esize] = ElemP[operand2, e, esize];
  else
    ElemP[result, e, esize] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BRKB

Break before first true condition

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |

BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<ZM> Is the predication qualifier, encoded in "M":

<table>
<thead>
<tr>
<th>M</th>
<th>&lt;ZM&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;
for e = 0 to elements-1
  boolean element = Elemp[operand, e, esize] == '1';
  if Elemp[mask, e, esize] == '1' then
    break = break || element;
  elseif merging then
    Elemp[result, e, esize] = if !break then '1' else '0';
  else
    Elemp[result, e, esize] = Elemp[operand2, e, esize];
  end
end
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
Break before first true condition, setting the condition flags

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
  boolean element = ElemP[operand, e, esize] == '1';
  ifElemP[mask, e, esize] == '1' then
    break = break || element;
    ElemP[result, e, esize] = if !break then '1' else '0';
  elsif merging then
    ElemP[result, e, esize] = ElemP[operand2, e, esize];
  else
    ElemP[result, e, esize] = '0';
  fi
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Propagate break to next partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Does not set the condition flags.

```
 0 0 1 0 0 1 0 1 0 0 1 1 0 0 0 0 1 5
```

**Assembler Symbols**

- `<Pdm>` Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[dm];
bits(PL) result;
if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros();
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm] = result;
```
Propagate break to next partition, setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | Pg | 0 | Pn | 0 | Pdm |


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;

Assembler Symbols

<Pdm> Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[dm];
bits(PL) result;
if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros();
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm] = result;
Break after first true condition, propagating from previous partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1'
        ElemP[result, e, 8] = if last then '1' else '0';
        last = last && (ElemP[operand2, e, 8] == '0');
    else
        ElemP[result, e, 8] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
Break after first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 1 0 1 0 0</td>
</tr>
</tbody>
</table>

**BRKPAS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B**

```c
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    ElemP[result, e, 8] = if last then '1' else '0';
    last = last && (ElemP[operand2, e, 8] == '0');
  else
    ElemP[result, e, 8] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```
BRKPB

Break before first true condition, propagating from previous partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

```
| 0 0 1 0 | 0 | 1 0 1 0 | 0 0 | Pm | 1 1 | Pg | 0 | Pn | 1 | Pd |
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler Symbols

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        last = last && (ElemP[operand2, e, 8] == '0');
        ElemP[result, e, 8] = if last then '1' else '0';
    else
        ElemP[result, e, 8] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BRKPBS

Break before first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |

S  

B


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd>  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>  Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>  Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>  Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');
for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    last = last & (ElemP[operand2, e, 8] == '0');
    ElemP[result, e, 8] = if last then '1' else '0';
  else
    ElemP[result, e, 8] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
CLASTA (scalar)

Conditionally extract element after last to general-purpose register

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 0 0 0 0 0 1 0 1 size 1 1 0 0 0 0 0 1 0 1 Pg Zm Rdn |

CLASTA <R><dn>, <Pg>, <R><dn>, <Zm><T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
integer csize = if esize < 64 then 32 else 64;
boolean isBefore = FALSE;

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x8</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = X[dn];
bits(VL) operand2 = Z[m];
bits(csize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = ZeroExtend(Elem[operand2, last, esize]);
    end
end
X[dn] = result;
CLASTA (SIMD&FP scalar)

Conditionally extract element after last to SIMD&FP scalar register

From the source vector register extract the element after the last active element, or if the last active element is the
final element extract element zero, and then zero-extend that element to destructively place in the destination and
first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the
least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

```
CLASTA <V><dn>, <Pg>, <V><dn>, <Zm>..<T>
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;
```

Assembler Symbols

- `<V>` Is a width specifier, encoded in "size":
  - `size <V>`
    - 00 B
    - 01 H
    - 10 S
    - 11 D

- `<dn>` Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the source scalable vector register, encoded in the "Zm" field.
- `<T>` Is the size specifier, encoded in "size":
  - `size <T>`
    - 00 B
    - 01 H
    - 10 S
    - 11 D
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(PL) mask = \texttt{P[g]};
bits(esize) operand1 = \texttt{V[dn]};
bits(VL) operand2 = \texttt{Z[m]};
bits(esize) result;
integer last = \texttt{LastActiveElement}(mask, esize);
if last < 0 then
  result = \texttt{ZeroExtend}(operand1);
else
  if !isBefore then
    last = last + 1;
    if last >= elements then last = 0;
    result = \texttt{Elem}[operand2, last, esize];
\texttt{V[dn]} = result;
CLASTA (vectors)

Conditionally extract element after last to vector register

From the second source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
```

CLASTA <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
    if last >= elements then last = 0;
    for e = 0 to elements - 1
        Elem[result, e, esize] = Elem[operand2, last, esize];
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CLASTB (scalar)

Conditionally extract last element to general-purpose register

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 1 0 1 | size | 1 1 0 0 0 | 1 1 0 1 | Pg | Zm | Rdn |

CLASTB <R><dn>, <Pg>, <R><dn>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
integer csize = if esize < 64 then 32 else 64;
boolean isBefore = TRUE;

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

< Pg > Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits[PL] mask = P[g];
bits(esize) operand1 = X[dn];
bits(VL) operand2 = Z[m];
bits(csize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = ZeroExtend(Elem[operand2, last, esize]);

X[dn] = result;
CLASTB (SIMD&FP scalar)

Conditionally extract last element to SIMD&FP scalar register

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Pg</td>
<td>Zm</td>
<td>Vdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CLASTB \(<V><dn>, <Pg>, <V><dn>, <Zm>.<T>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<dn>\) Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the source scalable vector register, encoded in the "Zm" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[dn];
bits(VL) operand2 = Z[m];
bits(esize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = Elem[operand2, last, esize];
V[dn] = result;
CLASTB (vectors)

Conditionally extract last element to vector register

From the second source vector register extract the last active element, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;
```

Assembler Symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
  result = operand1;
else
  if !isBefore then
    last = last + 1;
    if last >= elements then last = 0;
    for e = 0 to elements-1
      Elem[result, e, esize] = Elem[operand2, last, esize];
  Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CLS

Count leading sign bits (predicated)

Count leading sign bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

| size | Pg | Zn | Zd |

**CLS** <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = CountLeadingSignBits(element)<esize-1:0>;
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CLZ

Count leading zero bits (predicated)

Count leading zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & | & size & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & | & Pg & | & Zn & | & Zd
\end{array}
\]

CLZ \(<Zd>., <T>\), <Pg>/M, <Zn>..<T>

\[
\text{if } \text{!HaveSVE()} \text{ then UNDEFINED;}
\]

\[
\text{integer esize} = 8 \ll \text{UInt}(size);
\]

\[
\text{integer } g = \text{UInt}(Pg);
\]

\[
\text{integer } n = \text{UInt}(Zn);
\]

\[
\text{integer } d = \text{UInt}(Zd);
\]

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the “Zd” field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\(\text{CheckSVEEnabled}();\)

\[
\text{integer elements} = \text{VL DIV esize;}
\]

\[
\text{bits(PL) mask} = P[g];
\]

\[
\text{bits(VL) operand} = \text{if AnyActiveElement(mask, esize) then } Z[n] \text{ else Zeros();}
\]

\[
\text{bits(VL) result} = Z[d];
\]

for e = 0 to elements-1

\[
\text{if Elem}[\text{mask, e, esize}] == '1' \text{ then}
\]

\[
\text{bits(esize) element} = \text{Elem}[\text{operand, e, esize}];
\]

\[
\text{Elem}[\text{result, e, esize}] = \text{CountLeadingZeroBits(element)}<\text{esize}-1:0>;
\]

\[
Z[d] = \text{result};
\]

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CMP<cc> (immediate)**

Compare vector to immediate

Compare active integer elements in the source vector with an immediate, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS, LE, LO, LS, LT or NE. It has encodings from 10 classes: **Equal**, **Greater than**, **Greater than or equal**, **Higher**, **Higher or same**, **Less than**, **Less than or equal**, **Lower**, **Lower or same** and **Not equal**

### Equal

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | Pg | Zn | 0  | Pd |
| ne |

**CMPEQ** `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>`

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

### Greater than

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | Pg | Zn | 1  | Pd |
| It |

**CMPGT** `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>`

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

### Greater than or equal

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | Pg | Zn | 0  | Pd |
| It |

**CMPEQ** `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>`

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```
CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
integer imm = $Int(imm5);
boolean unsigned = FALSE;

Higher

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |    |

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Higher or same

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |    |

CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Less than

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |    |

CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
integer imm = $Int(imm5);
boolean unsigned = FALSE;
Less than or equal

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & \text{imm} & 0 & 0 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd}
\end{array}
\]

\text{CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>}

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer n = \text{UInt}(\text{Zn});
integer d = \text{UInt}(\text{Pd});
\text{SVECmp op} = \text{Cmp LE};
integer imm = \text{SInt}(\text{imm5});
boolean unsigned = FALSE;

Lower

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & \text{imm} & 1 & \text{Pg} & \text{Zn} & 0 & \text{Pd}
\end{array}
\]

\text{CMPLO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>}

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer n = \text{UInt}(\text{Zn});
integer d = \text{UInt}(\text{Pd});
\text{SVECmp op} = \text{Cmp LT};
integer imm = \text{UInt}(\text{imm7});
boolean unsigned = TRUE;

Lower or same

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & \text{imm} & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd}
\end{array}
\]

\text{CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>}

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer n = \text{UInt}(\text{Zn});
integer d = \text{UInt}(\text{Pd});
\text{SVECmp op} = \text{Cmp LE};
integer imm = \text{UInt}(\text{imm7});
boolean unsigned = TRUE;

Not equal

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & \text{imm} & 1 & 0 & 0 & \text{Pg} & \text{Zn} & 1 & \text{Pd}
\end{array}
\]

\text{CMP<cc> (immediate)}
CMPNE <$Pd> .<T> , <$Pg>/Z, <$Zn>.<T>, <$imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
integer imm = $Int(imm5);
boolean unsigned = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> For the equal, greater than, greater than or equal, less than, less than or equal and not equal variant: is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

For the higher, higher or same, lower and lower or same variant: is the unsigned immediate operand, in the range 0 to 127, encoded in the "imm7" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(PL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    boolean cond;
    case op of
      when Cmp_EQ cond = element1 == imm;
      when Cmp_NE cond = element1 != imm;
      when Cmp_GE cond = element1 >= imm;
      when Cmp_LT cond = element1 < imm;
      when Cmp_GT cond = element1 > imm;
      when Cmp_LE cond = element1 <= imm;
      ElemP[result, e, esize] = if cond then '1' else '0';
    else
      ElemP[result, e, esize] = '0';
  else
    ElemP[result, e, esize] = '0';
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
CMP<cc> (vectors)

Compare vectors

Compare active integer elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS or NE. This instruction is used by the pseudo-instructions CMPLE (vectors), CMPLO (vectors), CMPLS (vectors), and CMPLT (vectors).

It has encodings from 6 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same and Not equal

Equal

```
ne 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 | size | 0 | Zm | 1 | 0 | 1 | Pg | Zn | 0 | Pd |
```

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;

Greater than

```
ne 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 | size | 0 | Zm | 1 | 0 | 0 | Pg | Zn | 1 | Pd |
```

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;

Greater than or equal

```
ne 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 | size | 0 | Zm | 1 | 0 | 0 | Pg | Zn | 0 | Pd |
```

CMP<cc> (vectors)
CMPGE \(<Pd>\)\(<T>\), \(<Pg>/\<Z>, \<Zn>\)\(<T>\), \(<Zm>\)\(<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = \(8 << \text{UInt}(\text{size})\);
integer g = \text{UInt}(\text{Pg})
integer n = \text{UInt}(\text{Zn})
integer m = \text{UInt}(\text{Zm})
integer d = \text{UInt}(\text{Pd})
\text{SVECmp op = Cmp_GE;}
\text{boolean unsigned = FALSE;}

Higher

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|
| 0 0 1 0 0 1 0 0 | size | 0 | Zm | 0 | 0 | 0 | Pg | Zn | 1 | Pd |

CMPHI \(<Pd>\)\(<T>\), \(<Pg>/\<Z>, \<Zn>\)\(<T>\), \(<Zm>\)\(<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = \(8 << \text{UInt}(\text{size})\);
integer g = \text{UInt}(\text{Pg})
integer n = \text{UInt}(\text{Zn})
integer m = \text{UInt}(\text{Zm})
integer d = \text{UInt}(\text{Pd})
\text{SVECmp op = Cmp_GT;}
\text{boolean unsigned = TRUE;}

Higher or same

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|
| 0 0 1 0 0 1 0 0 | size | 0 | Zm | 0 | 0 | 0 | Pg | Zn | 0 | Pd |

CMPHS \(<Pd>\)\(<T>\), \(<Pg>/\<Z>, \<Zn>\)\(<T>\), \(<Zm>\)\(<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = \(8 << \text{UInt}(\text{size})\);
integer g = \text{UInt}(\text{Pg})
integer n = \text{UInt}(\text{Zn})
integer m = \text{UInt}(\text{Zm})
integer d = \text{UInt}(\text{Pd})
\text{SVECmp op = Cmp_GE;}
\text{boolean unsigned = TRUE;}

Not equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|
| 0 0 1 0 0 1 0 0 | size | 0 | Zm | 1 | 0 | 1 | Pg | Zn | 1 | Pd |

CMPNE \(<Pd>\)\(<T>\), \(<Pg>/\<Z>, \<Zn>\)\(<T>\), \(<Zm>\)\(<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = \(8 << \text{UInt}(\text{size})\);
integer g = \text{UInt}(\text{Pg})
integer n = \text{UInt}(\text{Zn})
integer m = \text{UInt}(\text{Zm})
integer d = \text{UInt}(\text{Pd})
\text{SVECmp op = Cmp_NE;}
\text{boolean unsigned = FALSE;}

\text{CMP<cc> (vectors)}
**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(PL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        case op of
            when Cmp_EQ cond = element1 == element2;
            when Cmp_NE cond = element1 != element2;
            when Cmp_GE cond = element1 >= element2;
            when Cmp_LT cond = element1 < element2;
            when Cmp_GT cond = element1 > element2;
            when Cmp_LE cond = element1 <= element2;
            ElemP[result, e, esize] = if cond then '1' else '0';
        else
            ElemP[result, e, esize] = '0';
        end;
    end;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMP<cc> (wide elements)

Compare vector to 64-bit wide elements

Compare active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS, LE, LO, LS, LT or NE. It has encodings from 10 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same, Less than, Less than or equal, Lower, Lower or same and Not equal

Equal

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 0 & 1 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 0 & 1 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
\end{array}
\]

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;

Greater than

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;

Greater than or equal

\[
\begin{array}{cccccccccccccccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
\end{array}
\]

CMP<cc> (wide elements)
CMPGE <Pd>,<T>, <Pg>/Z, <Zn>,<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;

Higher

\[
\begin{array}{cccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 1 & 1 & 0 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

CMPHI <Pd>,<T>, <Pg>/Z, <Zn>,<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;

Higher or same

\[
\begin{array}{cccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 1 & 1 & 0 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
\end{array}
\]

CMPHS <Pd>,<T>, <Pg>/Z, <Zn>,<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;

Less than

\[
\begin{array}{cccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 1 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
\end{array}
\]
CMPLT `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```c
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = FALSE;

---

Less than or equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|---------------|---------------|
| 0  0  1  0  1  0  0  | size 0         | 0  1  1       | Pg            | Zn            |
| Uit                                     |               |               |               |

CMPLE `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```c
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = FALSE;

---

Lower

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|---------------|---------------|
| 0  0  1  0  1  0  0  | size 0         | 1  1  1       | Pg            | Zn            |
| Uit                                     |               |               |               |

CMPLO `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```c
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = TRUE;

---

Lower or same

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|---------------|---------------|
| 0  0  1  0  1  0  0  | size 0         | 1  1  1       | Pg            | Zn            |
| Uit                                     |               |               |               |
CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = TRUE;

Not equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 0 0 1 0 0 1 0 0 | size | 0 | Zm | 0 0 | 1 | Pg | Zn | 1 | Pd |

CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(PL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, (e * esize) DIV 64, 64], unsigned);
  case op of
    when Cmp_EQ cond = element1 == element2;
    when Cmp_NE cond = element1 != element2;
    when Cmp_GE cond = element1 >= element2;
    when Cmp_LT cond = element1 < element2;
    when Cmp_GT cond = element1 > element2;
    when Cmp_LE cond = element1 <= element2;
  else
    ElemP[result, e, esize] = if cond then '1' else '0';
  end case
  ElemP[result, e, esize] = if cond then '1' else '0';
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
CMPLE (vectors)

Compare signed less than or equal to vector, setting the condition flags

Compare active signed integer elements in the first source vector being less than or equal to corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

### Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

### Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.
CMPLO (vectors)

Compare unsigned lower than vector, setting the condition flags

Compare active unsigned integer elements in the first source vector being lower than corresponding unsigned
elements in the second source vector, and place the boolean results of the comparison in the corresponding elements
of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N),
NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 0 | size | 0 | Zm | 0 | 0 | 0 | Pg | Zn | 1 | Pd |

CMPLO <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.
CMPLS (vectors)

Compare unsigned lower or same as vector, setting the condition flags

Compare active unsigned integer elements in the first source vector being lower than or same as corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of **CMP<cc> (vectors)**. This means:

- The encodings in this description are named to match the encodings of **CMP<cc> (vectors)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **CMP<cc> (vectors)** gives the operational pseudocode for this instruction.

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of **CMP<cc> (vectors)** gives the operational pseudocode for this instruction.
CMPLT (vectors)

Compare signed less than vector, setting the condition flags

Compare active signed integer elements in the first source vector being less than corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of \texttt{CMP<cc> (vectors)}. This means:

- The encodings in this description are named to match the encodings of \texttt{CMP<cc> (vectors)}.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of \texttt{CMP<cc> (vectors)} gives the operational pseudocode for this instruction.

\begin{center}
\begin{tabular}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & | & 0 & 1 & 0 & 0 & | & Pg & | & Zn & | & 1 & | & Pd & \hline
\end{tabular}
\end{center}

\texttt{CMPLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>}

is equivalent to

\texttt{CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>}

Assembler Symbols

\begin{itemize}
\item \texttt{<Pd>} is the name of the destination scalable predicate register, encoded in the "Pd" field.
\item \texttt{<Zm>} is the name of the second source scalable vector register, encoded in the "Zm" field.
\item \texttt{<Zn>} is the name of the first source scalable vector register, encoded in the "Zn" field.
\item \texttt{<T>} is the size specifier, encoded in "size":
\begin{center}
\begin{tabular}{c|c}
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\end{tabular}
\end{center}
\item \texttt{<Pg>} is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\end{itemize}

Operation

The description of \texttt{CMP<cc> (vectors)} gives the operational pseudocode for this instruction.
CNOT

Logically invert boolean condition in vector (predicated)

Logically invert the boolean value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Boolean TRUE is any non-zero value in a source, and one in a result element. Boolean FALSE is always zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 0 0 0 0 1 0 0 | size             | 0 1 1 | 0 1 1 1 0 1 | Pg   | Zn   | Zd   |

CNOT <Zd>,<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = ZeroExtend(IsZeroBit(element), esize);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CNT

Count non-zero bits (predicated)

Count non-zero bits in each active element of the source vector, and place the results in the corresponding elements of
the destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

CNT <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = BitCount(element)<esize-1:0>;
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CNTB, CNTD, CNTH, CNTW

Set scalar to multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate
in the range 1 to 16 inclusive, and then places the result in the scalar destination.
The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).
Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.
It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

Byte

CNTB <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Doubleword

CNTD <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Halfword

CNTH <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
CNTW <Xd>{, <pattern>{, MUL #<imm>}}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);

X[d] = (count * imm)<63:0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CNTP

Set scalar to count of true predicate elements

Counts the number of active and true elements in the source predicate and places the scalar result in the destination general-purpose register. Inactive predicate elements are not counted.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

CNTP <Xd>, <Pg>, <Pn>.<T>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Rd);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(64) sum = Zeros();
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && ElemP[operand, e, esize] == '1' then
        sum = sum + 1;
X[d] = sum;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**COMPACT**

Shuffle active elements of vector to the right and fill with zero

Read the active elements from the source vector and pack them into the lowest-numbered elements of the destination vector. Then set any remaining elements of the destination vector to zero.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>1 0 0 0 0 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**COMPACT** <Zd>.<T>, <Pg>, <Zn>.<T>

```c
if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UINT(size);
integer g = UINT(Pg);
integer n = UINT(Zn);
integer d = UINT(Zd);
```

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>S D</td>
</tr>
</tbody>
</table>
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Zeros();
integer x = 0;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand1, e, esize];
    Elem[result, x, esize] = element;
    x = x + 1;
Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CPY (immediate, merging)

Copy signed integer immediate to vector elements (merging)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias MOV (immediate, predicated, merging).
This instruction is used by the pseudo-instruction FMOV (zero, predicated).

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) dest = Z[d];
bits(VL) result;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = imm<esize-1:0>;
  elsif merging then
    Elem[result, e, esize] = Elem[dest, e, esize];
  else
    Elem[result, e, esize] = Zeros();

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CPY (immediate, zeroing)**

Copy signed integer immediate to vector elements (zeroing)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is “#<simm8>, LSL #8”. However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as “#0, LSL #8”.

This instruction is used by the alias **MOV (immediate, predicated, zeroing)**.

<table>
<thead>
<tr>
<th>CPY Zd.&lt;T&gt;, Pg/Z, #&lt;imm&gt;{, &lt;shift&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>if !HaveSVE() then UNDEFINED;</td>
</tr>
<tr>
<td>if size:sh == '001' then UNDEFINED;</td>
</tr>
<tr>
<td>integer esize = 8 &lt;&lt; UInt(size);</td>
</tr>
<tr>
<td>integer g = UInt(Pg);</td>
</tr>
<tr>
<td>integer d = UInt(Zd);</td>
</tr>
<tr>
<td>boolean merging = FALSE;</td>
</tr>
<tr>
<td>integer imm = SInt(imm8);</td>
</tr>
<tr>
<td>if sh == '1' then imm = imm &lt;&lt; 8;</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the “Zd” field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register, encoded in the ”Pg” field.
- **<imm>** Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.
- **<shift>** Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) dest = Z[d];
bits(VL) result;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = imm<esize-1:0>;
  elsif merging then
    Elem[result, e, esize] = Elem[dest, e, esize];
  else
    Elem[result, e, esize] = Zeros();

Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CPY (scalar)**

Copy general-purpose register to vector elements (predicated)

Copy the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias **MOV (scalar, predicated)**.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | size | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | Pg | Rn | Zd |

**CPY <Zd>.<T>, <Pg>/M, <R><n|SP>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Rn);
integer d = UInt(Zd);

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<R>** Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>
- **<n|SP>** Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) result = Z[d];
if AnyActiveElement(mask, esize) then
    bits(64) operand1;
    if n == 31 then
        operand1 = SP[];
    else
        operand1 = X[n];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = operand1<esize-1:0>;
Z[d] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CPY (SIMD&FP scalar)**

Copy SIMD&FP scalar register to vector elements (predicated)

Copy the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias [MOV (SIMD&FP scalar, predicated)].

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

**CPY <Zd>, <T>, <Pg>/M, <V>n>**

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
ineger g = UInt(Pg);
ineger n = UInt(Vn);
ineger d = UInt(Zd);
```

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.

- **<T>** Is the size specifier, encoded in “size”:

```plaintext
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- **<V>** Is a width specifier, encoded in “size”:

```plaintext
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- **<n>** Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

**Operation**

```plaintext
CheckSVEEnabled();
ieger elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = if AnyActiveElement(mask, esize) then V[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
    if Elemp[mask, e, esize] == '1' then
        Elem[result, e, esize] = operand1;

Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CTERMEQ, CTERMNE

Compare and terminate loop

Detect termination conditions in serialized vector loops. Tests whether the comparison between the scalar source operands holds true and if not tests the state of the !LAST condition flag (C) which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.

The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:
* GE (N=0 & V=0): continue loop (compare failed and last element not selected);
* LT (N=0 & V=1): terminate loop (last element selected);
* LT (N=1 & V=0): terminate loop (compare succeeded);

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.

It has encodings from 2 classes: Equal and Not equal

Equal

<table>
<thead>
<tr>
<th>sz</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1</td>
<td>0 0 1 0 0 0</td>
<td>0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

CTERMEQ <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_EQ;

Not equal

<table>
<thead>
<tr>
<th>sz</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1</td>
<td>0 0 1 0 0 0</td>
<td>1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

CTERMNE <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_NE;

Assembler Symbols

\(<R>\) Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

\(<n>\) Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

\(<m>\) Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.
Operation

CheckSVEEnabled();
bits(esize) operand1 = X[n];
bits(esize) operand2 = X[m];
integer element1 = UInt(operand1);
integer element2 = UInt(operand2);
boolean term;

case op of
  when Cmp_EQ term = element1 == element2;
  when Cmp_NE term = element1 != element2;
if term then
  PSTATE.N = '1';
PSTATE.V = '0';
else
  PSTATE.N = '0';
PSTATE.V = (NOT PSTATE.C);
DECB, DECD, DECH, DECW (scalar)

Decrement scalar by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

### Byte

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--|
| 0 0 0 0 0 1 0 0 | 0 0 1 1 | imm4 | 1 1 1 0 0 1 | pattern | Rdn |

DEC <Xdn>{, <pattern>{, MUL #<imm>}}

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Doubleword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--|
| 0 0 0 0 0 1 0 0 | 1 1 1 | imm4 | 1 1 1 0 0 1 | pattern | Rdn |

DECD <Xdn>{, <pattern>{, MUL #<imm>}}

```java
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Halfword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--|
| 0 0 0 0 0 1 0 0 | 0 1 1 | imm4 | 1 1 1 0 0 1 | pattern | Rdn |

DECH <Xdn>{, <pattern>{, MUL #<imm>}}

```java
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(64) operand1 = X[dn];

X[dn] = operand1 - (count * imm);
DECD, DECH, DECW (vector)

Decrement vector by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword, Halfword and Word

Doubleword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 1 1 1 | imm4 | 1 1 0 0 0 1 | pattern | Zdn |

DECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Halfword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 0 1 1 1 | imm4 | 1 1 0 0 0 1 | pattern | Zdn |

DECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Word

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 1 0 1 1 | imm4 | 1 1 0 0 0 1 | pattern | Zdn |

DECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] - (count * imm);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
DECP (scalar)

Decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination.

<table>
<thead>
<tr>
<th>DECP</th>
<th>&lt;Xdn&gt;, &lt;Pm&gt;.&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) operand1 = X[dn];
bits(PL) operand2 = P[m];
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
X[dn] = operand1 - count;
DECP (vector)

Decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
0 0 1 0 0 1 0 1 | size | 1 0 1 1 0 | Pm | Zdn
  D
```

**DECP <Zdn>..<T>, <Pm>..<T>**

```java
if !HaveSVE() then UNDEFINED;
if size == ‘00’ then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size <T>
    - 00 RESERVED
    - 01 H
    - 10 S
    - 11 D
- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  Elem[result, e, esize] = Elem[operand1, e, esize] - count;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
DUP (immediate)

Broadcast signed immediate to vector elements (unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias MOV (immediate, unpredicated).

This instruction is used by the pseudo-instruction FMOV (zero, unpredicated).

**Assembler Symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<imm>`: Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
- `<shift>`: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":
  
<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

- `CheckSVEEnabled();`
- `bits(VL) result = Replicate(imm<esize-1:0>);`
- `Z[d] = result;`

---

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DUP (indexed)

Broadcast indexed element to vector (unpredicated)

Unconditionally broadcast the indexed source vector element into each element of the destination vector. This instruction is unpredicated.

The immediate element index is in the range of 0 to 63 (bytes), 31 (halfwords), 15 (words), 7 (doublewords) or 3 (quadwords). Selecting an element beyond the accessible vector length causes the destination vector to be set to zero. This instruction is used by the alias MOV (SIMD&FP scalar, unpredicated).

<table>
<thead>
<tr>
<th>Bits (7)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>x100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
<tr>
<td>10000</td>
<td>Q</td>
</tr>
</tbody>
</table>

if !HaveSVE() then UNDEFINED;
bits(7) imm = imm2:tsz;
integer esize;
integer index;
case tsz of
  when '00000' UNDEFINED;
  when '10000' esize = 128; index = UInt(imm<6:5>);
  when 'x1000' esize = 64;  index = UInt(imm<6:4>);
  when 'xx100' esize = 32;  index = UInt(imm<6:3>);
  when 'xxx10' esize = 16;  index = UInt(imm<6:2>);
  when 'xxxx1' esize = 8;   index = UInt(imm<6:1>);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in “tsz”:

<table>
<thead>
<tr>
<th>tsz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>x100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
<tr>
<td>10000</td>
<td>Q</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm>  Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (SIMD&amp;FP scalar, unpredicated)</td>
<td>BitCount(imm2:tsz) == 1</td>
</tr>
<tr>
<td>MOV (SIMD&amp;FP scalar, unpredicated)</td>
<td>BitCount(imm2:tsz) &gt; 1</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z{n};
bits(VL) result;
bits(esize) element;

if index >= elements then
    element = Zeros();
else
    element = Elem[operand1, index, esize];
result = Replicate(element);

Z[d] = result;
**DUP (scalar)**

Broadcast general-purpose register to vector elements (unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (scalar, unpredicated).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 0 0 0 0 0 0 1 0 1 | size 1 | 0 0 0 0 0 0 0 1 | 1 1 0 | Rn | Zd |

**DUP** <Zd>.<T>, <R><n|SP>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
```

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) operand;
if n == 31 then
    operand = SP[];
else
    operand = X[n];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = operand<esize-1:0>;
Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DUPM

Broadcast logical bitmask immediate to vector (unpredicated)

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This instruction is used by the alias MOV.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------------------|--|
| 0 0 0 0 0 1 0 1 1 1 0 0 0 0 | imm13 |
| Zd |

DUPM <Zd>,<T>, #<const>

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer d = UInt(Zd);
bits(esize) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 xxxxxx</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>0 00xxx</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0 10xxx</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0 110xxx</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 1110xx</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 11110x</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 111110</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 111111</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1 xxxxxxx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>SVEMoveMaskPreferred(imm13)</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(VL) result = Replicate(imm);
Z[d] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EON

Bitwise exclusive OR with inverted immediate (unpredicated)

Bitwise exclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of **EOR (immediate)**. This means:

- The encodings in this description are named to match the encodings of **EOR (immediate)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **EOR (immediate)** gives the operational pseudocode for this instruction.

This is equivalent to

**EOR <Zdn>.<T>, <Zdn>.<T>, #<const>**

is equivalent to

**EOR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)**

### Assembler Symbols

- **<Zdn>** is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<const>** is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

### Operation

The description of **EOR (immediate)** gives the operational pseudocode for this instruction.

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EOR (immediate)

Bitwise exclusive OR with immediate (unpredicated)

Bitwise exclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in
the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or
zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction `EON`.

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field
containing a rotated run of non-zero bits, encoded in the “imm13” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 64;
bits(VL) operand = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 EOR imm;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**EOR (predicates)**

Bitwise exclusive OR predicates

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias **NOT (predicate)**.

**EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B**

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT (predicate)</td>
<td>Pm == Pg</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EOR (vectors, predicated)

Bitwise exclusive OR vectors (predicated)

Bitwise exclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

\[
\text{EOR } \langle \text{Zdn} \rangle, \langle \text{T} \rangle, \langle \text{Pg} \rangle / \text{M}, \langle \text{Zdn} \rangle . \langle \text{T} \rangle, \langle \text{Zm} \rangle . \langle \text{T} \rangle
\]

if \! \text{HaveSVE}() \text{ then UNDEFINED;}
integer esize = 8 << \text{UInt}(size);
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Zdn);
integer m = \text{UInt}(Zm);

Assembler Symbols

\(\langle \text{Zdn} \rangle\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(\langle \text{T} \rangle\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(\langle \text{T} \rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(\langle \text{Pg} \rangle\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(\langle \text{Zm} \rangle\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\text{CheckSVEEnabled}();
integer elements = \text{VL} \text{ DIV esize};
bits(PL) mask = \text{P}[g];
bits(VL) operand1 = \text{Z}[dn];
bits(VL) operand2 = if \text{AnyActiveElement}(mask, esize) then \text{Z}[m] else \text{Zeros}();
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = \text{Elem}[operand1, e, esize];
    bits(esize) element2 = \text{Elem}[operand2, e, esize];
    if \text{ElemP}[mask, e, esize] == '1' then
        \text{Elem}[result, e, esize] = element1 \text{ EOR} element2;
    else
        \text{Elem}[result, e, esize] = \text{Elem}[operand1, e, esize];
\end{verbatim}

\text{Z[dn]} = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EOR (vectors, unpredicated)

Bitwise exclusive OR vectors (unpredicated)

Bitwise exclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
 0 0 0 0 1 1 0 0 | Zm
 0 0 1 1 0 0 | Zn
    | Zd
```

EOR <Zd>.D, <Zn>.D, <Zm>.D

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
Z[d] = operand1 EOR operand2;
```

EORS

Bitwise exclusive OR predicates, setting the condition flags

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias NOTS.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Pm</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTS</td>
<td>Pm == Pg</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
EORV

Bitwise exclusive OR reduction to scalar

Bitwise exclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |

EORV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(pl) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(esize) result = Zeros(esize);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    result = result EOR Elem[operand, e, esize];
V[d] = result;
Extract vector from pair of vectors

Copy the indexed byte up to the last byte of the first source vector to the bottom of the result vector, then fill the remainder of the result starting from the first byte of the second source vector. The result is placed destructively in the first source vector. This instruction is unpredicated.

An index that is greater than or equal to the vector length in bytes is treated as zero, leaving the destination and first source vector unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------|-------|---------|--------|
|   0 0 0 0 0 1 0 1 0 0 1          |imm8h| 0 0 0   |
| imm8l       | Zm    | Zdn     |
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer position = UInt(imm8h:imm8l);

**Assembler Symbols**

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<imm>`: Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8h:imm8l" fields.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

if position >= elements then
    position = 0;

position = position << 3;
bits(VL*2) concat = operand2 : operand1;
result = concat<(position+VL)-1:position>;

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point absolute difference (predicated)

Compute the absolute difference of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{c|cccc|cc|cccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & P_g & Z_m & Z_d n \\
\end{array}
\]

**FABD** \(<Z_d n>\), \(<T>\), \(<P_g>/M\), \(<Z_d n>\), \(<Z_m>\)</n>

if \(!\text{HaveSVE}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 \(<\text{UInt}(\text{size})\);
integer g = \text{UInt}(P_g);
integer dn = \text{UInt}(Z_d n);
integer m = \text{UInt}(Z_m);

**Assembler Symbols**

\(<Z_d n>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<P_g>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Z_m>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\text{PL}) mask = \text{P}[g];
bits(\text{VL}) operand1 = \text{Z}[dn];
bits(\text{VL}) operand2 = if \text{AnyActiveElement}(\text{mask}, \text{esize}) \text{then} \text{Z}[m] \text{else} \text{Zeros}();
bits(\text{VL}) result;

for e = 0 to elements-1
  bits(esize) element1 = \text{Elem}[\text{operand1}, e, \text{esize}];
  if \text{ElemP}[\text{mask}, e, \text{esize}] == '1' then
    bits(esize) element2 = \text{Elem}[\text{operand2}, e, \text{esize}];
    \text{Elem}[\text{result}, e, \text{esize}] = \text{FPAbs}(\text{FPSub}(\text{element1}, \text{element2}, \text{FPCR[]}));
  else
    \text{Elem}[\text{result}, e, \text{esize}] = \text{element1};

\text{Z}[dn] = \text{result};

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FABS

Floating-point absolute value (predicated)

Take the absolute value of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

FABS <Zd>, <T>, <Pg>/M, <Zn>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPAbs(element);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FAC<cc>

Floating-point absolute compare vectors

Compare active absolute values of floating-point elements in the first source vector with corresponding absolute values of elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

The <cc> symbol specifies one of the standard ARM condition codes: GE, GT, LE, or LT.

This instruction is used by the pseudo-instructions FACLE, and FACLT.

It has encodings from 2 classes: Greater than and Greater than or equal.

Greater than

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 | 0 | Zm | 1 | 1 | 1 | Pg | 0 | Zn | 1 | 1 | Pd |
```

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
```

Greater than or equal

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 | 0 | Zm | 1 | 1 | 0 | Pg | 0 | Zn | 1 | 1 | Pd |
```

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
```

Assembler Symbols

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<T>** Is the size specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Reserved</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(PL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    boolean res;
    case op of
      when Cmp_GE res = FPCompareGE(FPAbs(element1), FPAbs(element2), FPCR[]);
      when Cmp_GT res = FPCompareGT(FPAbs(element1), FPAbs(element2), FPCR[]);
      else ElemP[result, e, esize] = if res then '1' else '0';
    end case
  else
    ElemP[result, e, esize] = '0';
  end if
P[d] = result;
**FACLE**

Floating-point absolute compare less than or equal

Compare active absolute values of floating-point elements in the first source vector being less than or equal to corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of **FAC<cc>**. This means:

- The encodings in this description are named to match the encodings of **FAC<cc>**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **FAC<cc>** gives the operational pseudocode for this instruction.

![Encoding Table](image)

**FACLE** <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

**FACGE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

**Assembler Symbols**

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of **FAC<cc>** gives the operational pseudocode for this instruction.
FACTL

Floating-point absolute compare less than

Compare active absolute values of floating-point elements in the first source vector being less than corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of \textit{FAC<cc>}. This means:

- The encodings in this description are named to match the encodings of \textit{FAC<cc>}. 
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of \textit{FAC<cc>} gives the operational pseudocode for this instruction.

\texttt{FACTL <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>}

\textit{is equivalent to}

\texttt{FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>}

\textbf{Assembler Symbols}

- \texttt{<Pd>} Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- \texttt{<Zm>} Is the name of the second source scalable vector register, encoded in the "Zm" field.
- \texttt{<Zn>} Is the name of the first source scalable vector register, encoded in the "Zn" field.
- \texttt{<T>} Is the size specifier, encoded in “size”:

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{size} & \textbf{<T>} \\
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & D \\
\hline
\end{tabular}
\end{center}

- \texttt{<Pg>} Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\textbf{Operation}

The description of \textit{FAC<cc>} gives the operational pseudocode for this instruction.
FADD (immediate)

Floating-point add immediate (predicated)

Add an immediate to each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bis(esize) imm = if i1 == '0' then FPointFive('0') else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPAdd(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADD (vectors, predicated)

Floating-point add vector (predicated)

Add active floating-point elements of the second source vector to corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1 | 1 | 0 | 0 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | |
| size | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | |

FADD <Zdn>., <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADD (vectors, unpredicated)

Floating-point add vector (unpredicated)

Add all floating-point elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

FADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>   Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
Z[d] = result;
FADDA

Floating-point add strictly-ordered reduction, accumulating in scalar

Floating-point add a SIMD&FP scalar source and all active lanes of the vector source and place the result destructively in the SIMD&FP scalar source register. Vector elements are processed strictly in order from low to high, with the scalar source providing the initial value. Inactive elements in the source vector are ignored.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 | size 0 1 1 0 0 0 0 1 | Pg Zm Vdn |

FADDA $<V><dn>, <Pg>, <V><dn>, <Zm>$.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
exteger esize = 8 << UInt(size);
exteger g = UInt(Pg);
exteger dn = UInt(Vdn);
exteger m = UInt(Zm);

Assembler Symbols

$<V>$ Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<dn>$ Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

$<Pg>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$<Zm>$ Is the name of the source scalable vector register, encoded in the "Zm" field.

$<T>$ Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
exteger elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(esize) result = operand1;

for e = 0 to elements - 1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand2, e, esize];
    result = FPAdd(result, element, FPCR[]);
V[dn] = result;
Floating-point add recursive reduction to scalar

Floating-point add horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.

```
0 1 1 0 0 1 0 1 | size | 0 0 0 | 0 0 0 | 0 0 1 | Pg | Zn | Vd
```

```
FADDV <V><d>, <Pg>, <Zn>.<T>
```

If !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

- `<V>` is a width specifier, encoded in "size":

```
 size    <V>
00  RESERVED
01  H
10  S
11  D
```

- `<d>` is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the source scalable vector register, encoded in the "Zn" field.

- `<T>` is the size specifier, encoded in "size":

```
 size    <T>
00  RESERVED
01  H
10  S
11  D
```

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(esize) identity = FPZero('0');
V[d] = ReducePredicated(ReduceOp_FADD, operand, mask, identity);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point complex add with rotate (predicated)

Add the real and imaginary components of the active floating-point complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. Destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

```
0 1 1 0 0 1 0 0 | size | 0 0 0 0 0 | rot | 1 0 0 | Pg | Zm | Zdn
```

FCADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

- **<Zdn>** is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<const>** is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#90</td>
</tr>
<tr>
<td>1</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;

for p = 0 to pairs-1
  acc_r = Elem[operand1, 2 * p + 0, esize];
  acc_i = Elem[operand1, 2 * p + 1, esize];
  if ElemP[mask, 2 * p + 0, esize] == '1' then
    elt2_i = Elem[operand2, 2 * p + 1, esize];
    if sub_i then elt2_i = FPNeg(elt2_i);
    acc_r = FPAdd(acc_r, elt2_i, FPCR[]);
  if ElemP[mask, 2 * p + 1, esize] == '1' then
    elt2_r = Elem[operand2, 2 * p + 0, esize];
    if sub_r then elt2_r = FPNeg(elt2_r);
    acc_i = FPAdd(acc_i, elt2_r, FPCR[]);
  Elem[result, 2 * p + 0, esize] = acc_r;
  Elem[result, 2 * p + 1, esize] = acc_i;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCM<cc> (vectors)

Floating-point compare vectors

Compare active floating-point elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, or NE, with the addition of UO for an unordered comparison.

This instruction is used by the pseudo-instructions FCMLE (vectors), and FCMLT (vectors).

It has encodings from 5 classes: Equal, Greater than, Greater than or equal, Not equal and Unordered

Equal

```
0 1 1 0 0 1 0 1 | size 0 | Zm 0 1 1 | Pg 0 | Zn 1 | Pd 0
```

```
FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
```

Greater than

```
0 1 1 0 0 1 0 1 | size 0 | Zm 0 1 0 | Pg 0 | Zn 1 | Pd 0
```

```
FCMG T <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
```

Greater than or equal

```
0 1 1 0 0 1 0 1 | size 0 | Zm 0 1 0 | Pg 0 | Zn 1 | Pd 0
```

```
```
FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;

Not equal

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmph cmpl</td>
</tr>
</tbody>
</table>

FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;

Unordered

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>

FCMUO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_UN;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 RESERVED</td>
</tr>
<tr>
<td>01 H</td>
</tr>
<tr>
<td>10 S</td>
</tr>
<tr>
<td>11 D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(\texttt{VL}) operand1 = if \texttt{AnyActiveElement}(mask, esize) then \texttt{Z}[n] else \texttt{Zeros}();
bits(\texttt{VL}) operand2 = if \texttt{AnyActiveElement}(mask, esize) then \texttt{Z}[m] else \texttt{Zeros}();
bits(\texttt{PL}) result;

for e = 0 to elements-1
  if \texttt{ElemP}[mask, e, esize] == '1' then
    bits(esize) element1 = \texttt{Elem}[operand1, e, esize];
    bits(esize) element2 = \texttt{Elem}[operand2, e, esize];
    boolean res;
    case op of
      when \texttt{Cmp\_EQ} res = \texttt{FPCompareEQ}(element1, element2, FPCR[]);
      when \texttt{Cmp\_GE} res = \texttt{FPCompareGE}(element1, element2, FPCR[]);
      when \texttt{Cmp\_GT} res = \texttt{FPCompareGT}(element1, element2, FPCR[]);
      when \texttt{Cmp\_UN} res = \texttt{FPCompareUN}(element1, element2, FPCR[]);
      when \texttt{Cmp\_NE} res = \texttt{FPCompareNE}(element1, element2, FPCR[]);
      when \texttt{Cmp\_LT} res = \texttt{FPCompareGT}(element2, element1, FPCR[]);
      when \texttt{Cmp\_LE} res = \texttt{FPCompareGE}(element2, element1, FPCR[]);
    \texttt{ElemP}[result, e, esize] = if res then '1' else '0';
  else
    \texttt{ElemP}[result, e, esize] = '0';

\texttt{P}[d] = result;
**FCM<cc> (zero)**

Floating-point compare vector with zero

Compare active floating-point elements in the source vector with zero, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

The `<cc>` symbol specifies one of the standard ARM condition codes: EQ, GE, GT, LE, LT, or NE. It has encodings from 6 classes: **Equal**, **Greater than**, **Greater than or equal**, **Less than**, **Less than or equal** and **Not equal**

### Equal

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 1 0 0 1 0 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
<td>ne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0**

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
```

### Greater than

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 1 0 0 1 0 0 0 0 0 0 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>1</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
<td>ne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0**

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
```

### Greater than or equal

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 1 0 0 1 0 0 0 0 0 0 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>lt</td>
<td>ne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0**

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
```
Less than

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \text{P} & \text{Z} & 0 & \text{P} & \text{d} \\
\end{array}
\]

\[\text{eq} \quad \text{lt} \quad \text{ne} \]

FCMLT \(<\text{Pd}>, <\text{T}>, <\text{Pg}>/Z, <\text{Zn}>, <\text{T}>, \#0.0\)

\[
\text{if } !\text{HaveSVE()} \text{ then UNDEFINED;}
\]
\[
\text{if size == '00' then UNDEFINED;}
\]
\[
\text{integer esize = 8 << UInt(size);}
\]
\[
\text{integer g = UInt(Pg);}
\]
\[
\text{integer n = UInt(Zn);}
\]
\[
\text{integer d = UInt(Pd);}
\]
\[
\text{SVECmp op = Cmp_LT;}
\]

Less than or equal

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \text{P} & \text{Z} & 1 & \text{P} & \text{d} \\
\end{array}
\]

\[\text{eq} \quad \text{lt} \quad \text{ne} \]

FCMLE \(<\text{Pd}>, <\text{T}>, <\text{Pg}>/Z, <\text{Zn}>, <\text{T}>, \#0.0\)

\[
\text{if } !\text{HaveSVE()} \text{ then UNDEFINED;}
\]
\[
\text{if size == '00' then UNDEFINED;}
\]
\[
\text{integer esize = 8 << UInt(size);}
\]
\[
\text{integer g = UInt(Pg);}
\]
\[
\text{integer n = UInt(Zn);}
\]
\[
\text{integer d = UInt(Pd);}
\]
\[
\text{SVECmp op = Cmp_LE;}
\]

Not equal

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \text{P} & \text{Z} & 0 & \text{P} & \text{d} \\
\end{array}
\]

\[\text{eq} \quad \text{lt} \quad \text{ne} \]

FCMNE \(<\text{Pd}>, <\text{T}>, <\text{Pg}>/Z, <\text{Zn}>, <\text{T}>, \#0.0\)

\[
\text{if } !\text{HaveSVE()} \text{ then UNDEFINED;}
\]
\[
\text{if size == '00' then UNDEFINED;}
\]
\[
\text{integer esize = 8 << UInt(size);}
\]
\[
\text{integer g = UInt(Pg);}
\]
\[
\text{integer n = UInt(Zn);}
\]
\[
\text{integer d = UInt(Pd);}
\]
\[
\text{SVECmp op = Cmp_NE;}
\]

Assembler Symbols

\(<\text{Pd}>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<\text{T}>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zn}>\) Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bhits(PL) mask = P[g];
bhits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bhits(PL) result;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    boolean res;
    case op of
      when Cmp_EQ res = FPCompareEQ(element, 0<esize-1:0>, FPCR[]);
      when Cmp_GE res = FPCompareGE(element, 0<esize-1:0>, FPCR[]);
      when Cmp_GT res = FPCompareGT(element, 0<esize-1:0>, FPCR[]);
      when Cmp_NE res = FPCompareNE(element, 0<esize-1:0>, FPCR[]);
      when Cmp_LT res = FPCompareGT(0<esize-1:0>, element, FPCR[]);
      when Cmp_LE res = FPCompareGE(0<esize-1:0>, element, FPCR[]);
        ElemP[result, e, esize] = if res then '1' else '0';
    else
      ElemP[result, e, esize] = '0';
  else
    ElemP[result, e, esize] = '0';
P[d] = result;
FCMLA (indexed)

Floating-point complex multiply-add by indexed values with rotate

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

The complex numbers within the second source vector are specified using an immediate index which selects the same complex number position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the size of the complex number. This instruction is unpredicated.

It has encodings from 2 classes: Half-precision and Single-precision

**Half-precision**

```assembly
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

**Single-precision**

```assembly
FCMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the single-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the half-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 3, encoded in the "i2" field.

For the single-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
integer pairspersegment = 128 DIV (2 * esize);
bis(VL) operand1 = Z[n];
bis(VL) operand2 = Z[m];
bis(VL) operand3 = Z[da];
bis(VL) result;
for p = 0 to pairs-1
  segmentbase = p - (p MOD pairspersegment);
s = segmentbase + index;
  addend_r = Elem[operand3, 2 * p + 0, esize];
  addend_i = Elem[operand3, 2 * p + 1, esize];
  elt1_a = Elem[operand1, 2 * p + sel_a, esize];
  elt2_a = Elem[operand2, 2 * s + sel_a, esize];
  elt2_b = Elem[operand2, 2 * s + sel_b, esize];
  if neg_r then elt2_a = FPNeg(elt2_a);
  if neg_i then elt2_b = FPNeg(elt2_b);
  addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
  addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
  Elem[result, 2 * p + 0, esize] = addend_r;
  Elem[result, 2 * p + 1, esize] = addend_i;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLA (vectors)

Floating-point complex multiply-add with rotate (predicated)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | Zm | 0  | rot | Pg | Zn | Zda |

FCMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = Z[da];
bits(VL) result;

for p = 0 to pairs-1
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    if ElemP[mask, 2 * p + 0, esize] == '1' then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_a = Elem[operand2, 2 * p + sel_a, esize];
        if neg_r then elt2_a = FPNeg(elt2_a);
        addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    if ElemP[mask, 2 * p + 1, esize] == '1' then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_b = Elem[operand2, 2 * p + sel_b, esize];
        if neg_i then elt2_b = FPNeg(elt2_b);
        addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMLE (vectors)**

Floating-point compare less than or equal to vector

Compare active floating-point elements in the first source vector being less than or equal to corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccccccccccccccccccc}
\end{array}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>size</th>
<th>0</th>
<th>Zm</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Pg</th>
<th>Zn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
</table>

\[
\text{cmph} \quad \text{cmpl}
\]

**FCMLE** <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

**FCMGE** <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zn>.<T>

**Assembler Symbols**

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<T>** Is the size specifier, encoded in "size":

\[
\begin{array}{c|c}
\text{size} & \text{<T>} \\
00 & \text{RESERVED} \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMLT (vectors)**

Floating-point compare less than vector

Compare active floating-point elements in the first source vector being less than corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

**FCMLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>**

is equivalent to

**FCMG T <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>**
FCPY

Copy 8-bit floating-point immediate to vector elements (predicated)

Copy a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias FMOV (immediate, predicated).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0   0   0   0   1   1   1   0   1   1   0   1   1   0   1   1   0   1   1   0   1   1   0   1   1   0 |
| size | Pg  | imm8 | Zd  |

FCPY <Zd>.<T>, <Pg>/M, #<const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
bits(esize) imm = VFPExpandImm(imm8);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n\times16\times2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) result = Z[d];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm;
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVT

Floating-point convert precision (predicated)

Convert the size and precision of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.


**Half-precision to single-precision**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|-------------------------------|
| 0 1 1 0 0 1 0 1 | 1 | 0 | 0 | 0 1 0 | 0 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |
```

FCVT <Zd>.S, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;

**Half-precision to double-precision**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|-------------------------------|
| 0 1 1 0 0 1 0 1 | 1 | 1 | 0 | 0 | 0 1 0 | 0 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |
```

FCVT <Zd>.D, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 64;

**Single-precision to half-precision**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|-------------------------------|
| 0 1 1 0 0 1 0 1 | 1 | 0 | 0 | 0 1 0 | 0 | 0 1 0 | 1 | Pg | Zn | Zd |
```

FCVT <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 16;
Single-precision to double-precision

```
FCVT <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;
```

Double-precision to half-precision

```
FCVT <Zd>.H, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 16;
```

Double-precision to single-precision

```
FCVT <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
```

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[]);
    Elem[result, e, esize] = ZeroExtend(res);
  
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FCVTZS

Floating-point convert to signed integer, rounding toward zero (predicated)

Convert to the signed integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.

FCVTZ <Zd>.D, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FCVTZ <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FCVTZ <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FCVTZ <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

int_U

FCVTZ <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

ChecksSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
        Elem[result, e, esize] = Extend(res, unsigned);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FCVTZU**

Floating-point convert to unsigned integer, rounding toward zero (predicated)

Convert to the unsigned integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.


### Half-precision to 16-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCVTZU <Zd>.H, <Pg>/M, <Zn>.H**

if !HaveSVE() then UNDEFINED;

integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;

boolean unsigned = TRUE;

FPRounding rounding = FPRounding_ZERO;

### Half-precision to 32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCVTZU <Zd>.S, <Pg>/M, <Zn>.H**

if !HaveSVE() then UNDEFINED;

integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;

boolean unsigned = TRUE;

FPRounding rounding = FPRounding_ZERO;

### Half-precision to 64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FCVTZU**
FCVTZU <Zd>.D, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | Pg | Zn | Zd |

int_U

FCVTZU <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | Pg | Zn | Zd |

int_U

FCVTZU <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | Pg | Zn | Zd |

int_U
FCVTZU <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | Pg | Zn | Zd |

int_U

FCVTZU <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

ChecksSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand  = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
    Elem[result, e, esize] = Extend(res, unsigned);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDIV

Floating-point divide by vector (predicated)

Divide active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 | size 0 0 1 1 0 1 1 0 0 | Pg | Zm | Zdn

FDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPDIV(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDIVR

Floating-point reversed divide by vector (predicated)

Reversed divide active floating-point elements of the second source vector by corresponding floating-point elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | Pg | Zm | Zdn |

FDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !\(\text{HaveSVE}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if \(\text{AnyActiveElement}(\text{mask}, \text{esize})\) then Z[m] else \(\text{Zeros}()\);
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element2, element1, FPCR[{}]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDUP

Broadcast 8-bit floating-point immediate to vector elements (unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias FMOV (immediate, unpredicated).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------------------|-----|
| size | imm8 | Zd |

FDUP `<Zd>`, `<T>`, `<const>`

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Zd);
bites(esize) imm = VFPExpandImm(imm8);
```

Assembler Symbols

`<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.

`<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

`<const>` Is a floating-point immediate value expressable as ±n×16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bites(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = imm;
Z[d] = result;
```
FEXPA

Floating-point exponential accelerator

The FEXPA instruction accelerates the polynomial series calculation of the \( \text{EXP}(x) \) function. The double-precision variant copies the low 52 bits of an entry from a hard-wired table of 64-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 11 bits of the source element (src<16:6>), setting the sign bit to zero. The single-precision variant copies the low 23 bits of an entry from hard-wired table of 32-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 8 bits of the source element (src<13:6>), setting the sign bit to zero. The half-precision variant copies the low 10 bits of an entry from hard-wired table of 16-bit coefficients, indexed by the low 5 bits of each element of the source vector, and prepends to that the next 5 bits of the source element (src<9:5>), setting the sign bit to zero.

A coefficient table entry with index \( M \) holds the floating-point value \( 2^{(m/64)} \), or for the half-precision variant \( 2^{(m/32)} \). This instruction is unpredicated.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| size | 1 0 0 0 0 0 | Zn | Zd |

**FEXPA** <Zd>, <T>, <Zn>, <T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

CheckSVEEnabled(); integer elements = VL DIV esize;
bits(VL) operand = Z[n];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPExpA(element);
Z[d] = result;
FMAD

Floating-point fused multiply-add vectors (predicated), writing multiplicand \([Z_{dn} = Z_a + Z_{dn} \times Z_m]\)

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{ccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & Z_a & 1 & 0 & 0 & P_g & Z_m & Z_{dn} & \text{N o p}
\end{array}
\]

FMAD <Zdn>.<T>, <P_g>/M, <Zm>.<T>, <Za>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 \ll \text{UInt}(size);
integer g = \text{UInt}(P_g);
integer dn = \text{UInt}(Z_{dn});
integer m = \text{UInt}(Z_m);
integer a = \text{UInt}(Z_a);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<P_g> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

\[
\text{CheckSVEEnabled}();
\]
integer elements = \text{VL} \div \text{esize};
bits(P_l) mask = P[g];
bits(V_l) operand1 = Z[dn];
bits(V_l) operand2 = if \text{AnyActiveElement}(mask, esize) then Z[m] else \text{Zeros}();
bits(V_l) operand3 = if \text{AnyActiveElement}(mask, esize) then Z[a] else \text{Zeros}();
bits(V_l) result;
for e = 0 to elements-1
  if \text{Elem}[mask, e, esize] == '1' then
    bits(esize) element1 = \text{Elem}[operand1, e, esize];
    bits(esize) element2 = \text{Elem}[operand2, e, esize];
    bits(esize) element3 = \text{Elem}[operand3, e, esize];
    if op1_neg then element1 = \text{FPNeg}(element1);
    if op3_neg then element3 = \text{FPNeg}(element3);
    \text{Elem}[result, e, esize] = \text{FPMulAdd}(element3, element1, element2, FPCR[]);
  else
    \text{Elem}[result, e, esize] = \text{Elem}[operand1, e, esize];
\]
\[Z_{dn} = \text{result};\]
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (immediate)

Floating-point maximum with immediate (predicated)

Determine the maximum of an immediate and each active floating-point element of the source vector, and
destructively place the results in the corresponding elements of the source vector. The immediate may take the value
+0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector
register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| 0  1  1  0  0  1  0  1 | size          | 0  1  1  1  1  0  1  0  0 | Pg          | 0  0  0  0  i1 | Zdn         |

FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMax(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
• The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The `MOVPRFX` instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAX (vectors)

Floating-point maximum (predicated)

Determine the maximum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

Assmblr Symols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMAX(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNM (immediate)

Floating-point maximum number with immediate (predicated)

Determine the maximum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is a quiet NaN, then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

```
0 1 1 0 0 1 0 1 | size  | 0 1 1 1 0 0 1 0 0 | Pg  | 0 0 0 0 | i1 | Zdn
```

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem(operand1, e, esize);
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMaxNum(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNM (vectors)

Floating-point maximum number (predicated)

Determine the maximum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is NaN then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNMV

Floating-point maximum number recursive reduction to scalar

Floating-point maximum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

FMAXNMV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<s> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;s&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(esize) identity = FPDefaultNaN();

V[d] = ReducePredicated(ReduceOp_FMAXNUM, operand, mask, identity);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXV

Floating-point maximum recursive reduction to scalar

Floating-point maximum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as -Infinity.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \text{P}g & \text{Z}n & \text{V}d
\end{array}
\]

FMAXV <V><d>, <Pg>, <Zn>.<T>

if !\texttt{HaveSVE}() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer g = \texttt{UInt}(Pg);
integer n = \texttt{UInt}(Zn);
integer d = \texttt{UInt}(Vd);

Assembler Symbols

\(<V>\) Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

\texttt{CheckSVEEnabled}();
bits(PL) mask = P[g];
bits(VL) operand = if \texttt{AnyActiveElement}(mask, esize) then Z[n] else \texttt{Zeros}();
bits(esize) identity = \texttt{FPInfinity}('1');
\texttt{V}[d] = \texttt{ReducePredicated}(\texttt{ReduceOp_FMAX}, operand, mask, identity);
FMIN (immediate)

Floating-point minimum with immediate (predicated)

Determine the minimum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  1  1  0  0  1  0  1</td>
</tr>
</tbody>
</table>

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
ninteger elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMin(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMIN (vectors)

Floating-point minimum (predicated)

Determine the minimum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If the element value is a quiet NaN, then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMINNM (immediate)**

Floating-point minimum number with immediate (predicated)

Determine the minimum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**FMINNM**<Zdn>**.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 &< UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem(operand1, e, esize);
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMinNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The **MOVPRFX** instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMINNM (vectors)

Floating-point minimum number (predicated)

Determine the minimum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[)];
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMINNMV

Floating-point minimum number recursive reduction to scalar

Floating-point minimum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|--------|--------|--------|--------|--------|--------|
| 0 1 1 0 0 1 0 1  | size | 0 0 0 | 1 0 1 0 0 1 | Pg | Zn | Vd |

FMINNMV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(esize) identity = FPDefaultNaN();

V[d] = ReducePredicated(ReduceOp_FMINNUM, operand, mask, identity);

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINV

Floating-point minimum recursive reduction to scalar

Floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity.

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

### Operation

- `CheckSVEEnabled();`
- `bits(PL) mask = P[g];`
- `bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();`
- `bits(esize) identity = FPInfinity('0');`
- `V[d] = ReducePredicated(ReduceOp_FMIN, operand, mask, identity);`

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (indexed)

Floating-point fused multiply-add by indexed elements \((Zda = Zda + Zn \times Zm[\text{indexed}])\)

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision and Double-precision.

**Half-precision**

```
0 1 1 0 0 1 0 0 0 13h 13l Zm 0 0 0 0 0 0 Zn Zda
```

```
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

**Single-precision**

```
0 1 1 0 0 1 0 0 1 0 1 i2 Zm 0 0 0 0 0 0 Zn Zda
```

```
FMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>]}
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

**Double-precision**

```
0 1 1 0 0 1 0 0 1 1 i1 Zm 0 0 0 0 0 0 Zn Zda
```

```
FMLA (indexed)
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(il);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
<imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.
For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decr3, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (vectors)

Floating-point fused multiply-add vectors (predicated), writing addend \(Z_{da} = Z_{da} + Z_{n} \times Z_{m}\)

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | size | 1 | Zm | 0 | 0 | 0 | Pg | Zn | Zda |
```

FMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];
  Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLS (indexed)

Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[indexed])

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision and Double-precision

Half-precision

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 1 0 0 1 0 0 | 0 | i3h | i3l | Zm | 0 0 0 0 0 | 1 | Zn | Zda |
```


if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Single-precision

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 1 0 0 1 0 0 | 1 | 0 | i2 | Zm | 0 0 0 0 0 | 1 | Zn | Zda |
```

FMLS <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Double-precision

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 1 0 0 1 0 0 | 1 | 1 | i1 | Zm | 0 0 0 0 0 | 1 | Zn | Zda |
```

FMLS (indexed)

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(il);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Assembler Symbols

<Zda>  Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn>   Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm>   For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
       For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
<imm>  For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
       For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.
       For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, s, esize];
  bits(esize) element3 = Elem[result, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  if op3_neg then element3 = FPNeg(element3);
  Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLS (vectors)

Floating-point fused multiply-subtract vectors (predicated), writing addend \[Z_{da} = Z_{da} + -Z_n \times Z_m\]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

\[\text{CheckSVEEnabled}();\]
\[\text{integer elements} = \text{VL \ div \ esize};\]
\[\text{bits(PL) mask} = \text{P}[g];\]
\[\text{bits(VL) operand1} = \text{if AnyActiveElement(mask, esize) then Z}[n] \text{ else Zeros();}\]
\[\text{bits(VL) operand2} = \text{if AnyActiveElement(mask, esize) then Z}[m] \text{ else Zeros();}\]
\[\text{bits(VL) operand3} = Z[da];\]
\[\text{bits(VL) result};\]
\[\text{for } e = 0 \text{ to elements-1 } \]
\[\text{if } \text{Elem}[\text{mask, e, esize}] = '1' \text{ then}\]
\[\text{bits(esize) element1} = \text{Elem}[\text{operand1, e, esize}];\]
\[\text{bits(esize) element2} = \text{Elem}[\text{operand2, e, esize}];\]
\[\text{bits(esize) element3} = \text{Elem}[\text{operand3, e, esize}];\]
\[\text{if op1_neg then } \text{element1} = \text{FPNeg(element1)};\]
\[\text{if op2_neg then } \text{element2} = \text{FPNeg(element2)};\]
\[\text{Elem}[\text{result, e, esize}] = \text{FPMulAdd(element3, element1, element2, FPCR[]);}\]
\[\text{else}\]
\[\text{Elem}[\text{result, e, esize}] = \text{Elem}[\text{operand3, e, esize}];\]
\[Z[da] = \text{result};\]
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMMLA

Floating-point matrix multiply-accumulate

The floating-point matrix multiply-accumulate instruction supports single-precision and double-precision data types in a \(2 \times 2\) matrix contained in segments of 128 or 256 bits, respectively. It multiplies the \(2 \times 2\) matrix in each segment of the first source vector by the \(2 \times 2\) matrix in the corresponding segment of the second source vector. The resulting \(2 \times 2\) matrix product is then destructively added to the matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F32MM indicates whether the single-precision variant is implemented.
ID_AA64ZFR0_EL1.F64MM indicates whether the double-precision variant is implemented.
It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element
(FEAT_F32MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | Zm | 1  | 1  | 1  | 0  | 0  | 1  | Zn | 1  | 1  | 1  | 0  | 0  | 1  | Zda |

FMMLA \(<\text{Zda}>.S, <\text{Zn}>.S, <\text{Zm}>.S>

if \(!\text{HaveSVEFP32MatMulExt}()\) then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit element
(FEAT_F64MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 1  | 1  | 1  | 0  | 0  | 1  | Zn | 1  | 1  | 1  | 0  | 0  | 1  | Zda |

FMMLA \(<\text{Zda}>.D, <\text{Zn}>.D, <\text{Zm}>.D>

if \(!\text{HaveSVEFP64MatMulExt}()\) then UNDEFINED;
integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

\(<\text{Zda}>\) Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
\(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the “Zn” field.
\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the “Zm” field.
Operation

```plaintext
CheckSVEEnabled();
if (VL < esize * 4) then UNDEFINED;
integer segments = VL DIV (4 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(4*esize) op1, op2;
bits(4*esize) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 4*esize];
    op2 = Elem[operand2, s, 4*esize];
    addend = Elem[operand3, s, 4*esize];
    res = FPMatMulAdd(addend, op1, op2, esize, FPCR[]);
    Elem[result, s, 4*esize] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (immediate, predicated)

Move 8-bit floating-point immediate to vector elements (predicated)

Move a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of FCPY. This means:

• The encodings in this description are named to match the encodings of FCPY.
• The description of FCPY gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 0 | 1 | Pg | 1 | 1 | 0 | imm8 | Zd |

FMOV <Zd>,<T>, <Pg>/M, #<const>

is equivalent to

FCPY <Zd>,<T>, <Pg>/M, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n÷16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FCPY gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPREFIX instruction. The MOVPREFIX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREFIX and this instruction is UNPREDICTABLE:

• The MOVPREFIX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPREFIX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand and register of this instruction.
FMOV (immediate, unpredicated)

Move 8-bit floating-point immediate to vector elements (unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This is an alias of FDUP. This means:

- The encodings in this description are named to match the encodings of FDUP.
- The description of FDUP gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | imm8| Zd |

FMOV <Zd>,<T>, #<const>

is equivalent to

FDUP <Zd>,<T>, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a floating-point immediate value expressable as ±n+16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FDUP gives the operational pseudocode for this instruction.
FMOV (zero, predicated)

Move floating-point +0.0 to vector elements (predicated)

Move floating-point constant +0.0 to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is a pseudo-instruction of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
</tr>
<tr>
<td>M sh</td>
</tr>
</tbody>
</table>

FMOV <Zd>.<T>, <Pg>/M, #0.0

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

| size | <T> |
|-------------------|
| 00 | RESERVED |
| 01 | H |
| 10 | S |
| 11 | D |

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMOV (zero, unpredicated)

Move floating-point +0.0 to vector elements (unpredicated)

Unconditionally broadcast the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This is a pseudo-instruction of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of DUP (immediate) gives the operational pseudocode for this instruction.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.
Floating-point fused multiply-subtract vectors (predicated), writing multiplicand \( Z_{dn} = Z_a + -Z_{dn} \times Z_m \)

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

### Assembler Symbols

- **<Zdn>** is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** is the size specifier, encoded in "size"

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Za>** is the name of the third source scalable vector register, encoded in the "Za" field.

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = <Z[dn]>
bits(VL) operand2 = if AnyActiveElement(mask, esize) then <Z[m]> else Zeros();
bits(VL) operand3 = if AnyActiveElement(mask, esize) then <Z[a]> else Zeros();
bits(VL) result;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMUL (immediate)**

Floating-point multiply by immediate (predicated)

Multiply by an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +2.0 only. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>i1</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>**

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bite(size) imm = if i1 == '0' then FPointFive('0') else FPTwo('0');

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#2.0</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bite(PL) mask = P[g];
bite(VL) operand1 = Z[dn];
bite(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMul(element1, imm, FPCR[]); 
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

• The **MOVPRFX** instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMUL (indexed)

Floating-point multiply by indexed elements

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision and Double-precision

### Half-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 1 1 0 0 1 0 0 | 0 | i3h | i3l | Zm | 0 0 1 0 0 0 | Zn | Zd |


if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### Single-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 1 1 0 0 1 0 0 | 1 | 0 | 1 | i2 | Zm | 0 0 1 0 0 0 | Zn | Zd |

FMUL <Zd>.S, <Zn>.S, <Zm>.S<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### Double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 1 1 0 0 1 0 0 | 1 | 1 | i1 | Zm | 0 0 1 0 0 0 | Zn | Zd |

FMUL <Zd>.D, <Zn>.D, <Zm>.D<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.

For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

---

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
Z[d] = result;
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMUL (vectors, predicated)**

Floating-point multiply vectors (predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | Zm | Zdn |

**FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>**

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<size> <T>

<table>
<thead>
<tr>
<th>00</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMUL (vectors, unpredicated)

Floating-point multiply vectors (unpredicated)

Multiply all elements of the first source vector by corresponding floating-point elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 0 0 1 0 1 | size | 0 | Zm | 0 0 0 0 1 0 | Zn | Zd |

FMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
Z[d] = result;
FMULX

Floating-point multiply-extended vectors (predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector except that ∞×0.0 gives 2.0 instead of NaN, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

The instruction can be used with FRECPX to safely convert arbitrary elements in mathematical vector space to UNIT VECTORS or DIRECTION VECTORS with length 1.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | | Pg | Zm | Zdn |

FMULX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FNEG**

Floating-point negate (predicated)

Negate each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This inverts the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 0 | size | 0 1 1 | 1 0 1 1 0 1 | Pg | Zn | Zd |

**FNEG** <Zd>, <T>, <Pg>/M, <Zn>.

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPNeg(element);
Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point negated fused multiply-add vectors (predicated), writing multiplicand \[Z_{dn} = -Z_a + -Z_{dn} \times Z_m\]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

**Assembler Symbols**

\(<Z_{dn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Z_m>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

\(<Z_a>\) Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FNMLA

Floating-point negated fused multiply-add vectors (predicated), writing addend \[Zda = -Zda + -Zn \times Zm\]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```assembly
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;
```

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in "size":
  ```
  size <T> 00 RESERVED 01 H 10 S 11 D
  ```
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```assembly
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = Elem[operand3, e, esize];
Z[da] = result;
```

Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FNMLS

Floating-point negated fused multiply-subtract vectors (predicated), writing addend \([Zda = -Zda + Zn \times Zm]\)

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 0  | 1  | 0  | 1  |    |

N op
```

FNMLS \(<\text{Zda}.<\text{T}>, \text{<Pg>}/\text{M}, \text{<Zn>.<T>}, \text{<Zm>.<T>}>\)

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer m = Uint(Zm);
integer da = Uint(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;
```

Assembler Symbols

\(<\text{Zda}>\) Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

\(<\text{T}>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

\texttt{CheckSVEEnabled();}
integer elements = \texttt{VL} \div \texttt{esize};
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(\texttt{VL}) operand1 = if \texttt{AnyActiveElement}(mask, esize) then \texttt{Z}[n] else \texttt{Zeros}();
bits(\texttt{VL}) operand2 = if \texttt{AnyActiveElement}(mask, esize) then \texttt{Z}[m] else \texttt{Zeros}();
bits(\texttt{VL}) operand3 = \texttt{Z}[da];
bits(\texttt{VL}) result;

for e = 0 to elements-1
  if \texttt{ElemP[mask, e, esize]} == '1' then
    bits(esize) element1 = \texttt{Elem}[operand1, e, esize];
    bits(esize) element2 = \texttt{Elem}[operand2, e, esize];
    bits(esize) element3 = \texttt{Elem}[operand3, e, esize];

    if \texttt{op1 neg} then element1 = \texttt{FPNeg}(element1);
    if \texttt{op3 neg} then element3 = \texttt{FPNeg}(element3);
    \texttt{Elem[result, e, esize]} = \texttt{FPMulAdd}(element3, element1, element2, FPCR[]);
  else
    \texttt{Elem[result, e, esize]} = \texttt{Elem}[operand3, e, esize];

\texttt{Z[da]} = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \texttt{MOVPRFX} instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand \([Z_{dn} = -Z_a + Z_{dn} \times Z_m]\)

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

```
0 1 1 0 0 1 0 1 | size | 1 | Za | 1 | 1 | Pg | Zm | Zdn
```

**FNMSB** `<Z_{dn}>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>`

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Z_{dn});
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;
```

**Assembler Symbols**

- `<Z_{dn}>` Is the name of the first source and destination scalable vector register, encoded in the "Z_{dn}" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Za>` Is the name of the third source scalable vector register, encoded in the "Za" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FRECPE

Floating-point reciprocal estimate (unpredicated)

Find the approximate reciprocal of each floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

FRECPE <Zd>.<T>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 * UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL / esize;
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRrecipEstimate(element, FPCR[]);
Z[d] = result;
FRECPS

Floating-point reciprocal step (unpredicated)

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 2.0 without intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.

\[
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{size} & \text{Zm} & \text{Zn} & \text{Zd} \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

FRECPS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRECPX

Floating-point reciprocal exponent (predicated)

Invert the exponent and zero the fractional part of each active floating-point element of the source vector, and place
the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register
remain unmodified.

The result of this instruction can be used with FMULX to convert arbitrary elements in mathematical vector space to
"unit vectors" or "direction vectors" of length 1.

FRECPX <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPREcpX(element, FPCR[]);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.
**FRINT<r>**

Floating-point round to integral value (predicated)

Round to an integral floating-point value with the specified rounding option from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

The `<r>` symbol specifies one of the following rounding options: N (to nearest, with ties to even), A (to nearest, with ties away from zero), M (toward minus Infinity), P (toward plus Infinity), Z (toward zero), I (current FPCR rounding mode), or X (current FPCR rounding mode, signalling inexact).

It has encodings from 7 classes: Current mode, Current mode signalling inexact, Nearest with ties to away, Nearest with ties to even, Toward zero, Toward minus infinity and Toward plus infinity

### Current mode

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 0 0 1 1 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**FRINTI <Zd>.<T>, <Pg>/M, <Zn>.<T>**

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### Current mode signalling inexact

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 0 0 1 1 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**FRINTX <Zd>.<T>, <Pg>/M, <Zn>.<T>**

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

### Nearest with ties to away

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 0 0 1 1 0 1 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**FRINTA <Zd>.<T>, <Pg>/M, <Zn>.<T>**

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
```
Nearest with ties to even

<table>
<thead>
<tr>
<th>Nearest with ties to even</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 0 0 1 0 1 1 size 0 0 0 0 0 0 1 0 1 Pg Zn Zd</td>
</tr>
</tbody>
</table>

FRINTN <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;

Toward zero

<table>
<thead>
<tr>
<th>Toward zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 0 0 1 0 1 1 size 0 0 0 0 1 1 1 0 1 Pg Zn Zd</td>
</tr>
</tbody>
</table>

FRINTZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_ZERO;

Toward minus infinity

<table>
<thead>
<tr>
<th>Toward minus infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 0 0 1 0 1 1 size 0 0 0 0 1 0 1 0 1 Pg Zn Zd</td>
</tr>
</tbody>
</table>

FRINTM <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;

Toward plus infinity

<table>
<thead>
<tr>
<th>Toward plus infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 1 1 0 0 1 0 1 1 size 0 0 0 0 0 1 1 0 1 Pg Zn Zd</td>
</tr>
</tbody>
</table>

FRINTP <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;
Assembler Symbols

\(<\text{Zd}\rangle\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<\text{T}\rangle\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{T}\rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}\rangle\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zn}\rangle\) Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\(\text{CheckSVEEnabled}();\)

\begin{verbatim}
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
Z[d] = result;
\end{verbatim}

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point reciprocal square root estimate (unpredicated)

Find the approximate reciprocal square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```每年都
  if !HaveSVE() then UNDEFINED;
  if size == '00' then UNDEFINED;
  integer esize = 8 << UInt(size);
  integer n = UInt(Zn);
  integer d = UInt(Zd);

  CheckSVEEnabled();
  integer elements = VL DIV esize;
  bits(VL) operand = Z[n];
  bits(VL) result;

  for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR[]);

  Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FRSQRTS**

Floating-point reciprocal square root step (unpredicated)

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 3.0 and divide the results by 2.0 without any intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**FRSQRTS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>**

```latex
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```latex
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);
Z[d] = result;
```

---

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FScale

Floating-point adjust exponent by vector (predicated)

Multiply the active floating-point elements of the first source vector by 2.0 to the power of the signed integer values in
the corresponding elements of the second source vector and destructively place the results in the corresponding
elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

FScale <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits([PL]) mask = P[g];
bits([VL]) operand1 = Z[dn];
bits([VL]) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits([VL]) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        integer element2 = SInt(Elem[operand2, e, esize]);
        Elem[result, e, esize] = FPScale(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSQRT

Floating-point square root (predicated)

Calculate the square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

FSQRT <Zd>, <T>, <Pg>/M, <Zn>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPSqrt(element, FPCR[]);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUB (immediate)

Floating-point subtract immediate (predicated)

Subtract an immediate from each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0') else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(element1, imm, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUB (vectors, predicated)

Floating-point subtract vectors (predicated)

Subtract active floating-point elements of the second source vector from corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

< Pg > Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits[PL] mask = P[g];
bits[VL] operand1 = Z[dn];
bits[VL] operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits[VL] result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPSub(element1, element2, FPCR());
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUB (vectors, unpredicated)

Floating-point subtract vectors (unpredicated)

Subtract all floating-point elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 0 0 1 0 1 | size 0 | Zm 0 0 0 0 0 1 | Zn | Zd |

FSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);  
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUBR (immediate)

Floating-point reversed subtract from immediate (predicated)

Reversed subtract from an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if ! HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPPointFive('0') else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(imm, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUBR (vectors)

Floating-point reversed subtract vectors (predicated)

Reversed subtract active floating-point elements of the first source vector from corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></th>
<th>Pg</th>
<th></th>
<th></th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
   if ElemP[mask, e, esize] == '1' then
      bits(esize) element2 = Elem[operand2, e, esize];
      Elem[result, e, esize] = FPSub(element2, element1, FPCR[]);
   else
      Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FTMAD

Floating-point trigonometric multiply-add coefficient

The FTMAD instruction calculates the series terms for either \( \sin(x) \) or \( \cos(x) \), where the argument \( x \) has been adjusted to be in the range \(-\pi/4 < x \leq \pi/4\).

To calculate the series terms of \( \sin(x) \) and \( \cos(x) \) the initial source operands of FTMAD should be zero in the first source vector and \( x^2 \) in the second source vector. The FTMAD instruction is then executed eight times to calculate the sum of eight series terms, which gives a result of sufficient precision.

The FTMAD instruction multiplies each element of the first source vector by the absolute value of the corresponding element of the second source vector and performs a fused addition of each product with a value obtained from a table of hard-wired coefficients, and places the results destructively in the first source vector.

The coefficients are different for \( \sin(x) \) and \( \cos(x) \), and are selected by a combination of the sign bit in the second source element and an immediate index in the range 0 to 7.

This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zm | Zdn |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<imm> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "imm3" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigMAdd(imm, element1, element2, FPCR[]);
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FTSMUL

Floating-point trigonometric starting value

The FTSMUL instruction calculates the initial value for the FTMAD instruction. The instruction squares each element in the first source vector and then sets the sign bit to a copy of bit 0 of the corresponding element in the second source register, and places the results in the destination vector. This instruction is unpredicated.

To compute $\sin(x)$ or $\cos(x)$ the instruction is executed with elements of the first source vector set to $x$, adjusted to be in the range $-\pi/4 < x \leq \pi/4$.

The elements of the second source vector hold the corresponding value of the quadrant number as an integer not a floating-point value. The value $q$ satisfies the relationship $(2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4$.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td>0</td>
<td>0 0 0 0 1 1</td>
<td>Zn</td>
<td>Zd</td>
</tr>
</tbody>
</table>

**Assemble Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```asm
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSMul(element1, element2, FPCR[]);
Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FTSSEL

Floating-point trigonometric select coefficient

The FTSSEL instruction selects the coefficient for the final multiplication in the polynomial series approximation. The instruction places the value 1.0 or a copy of the first source vector element in the destination element, depending on bit 0 of the quadrant number \( q \) held in the corresponding element of the second source vector. The sign bit of the destination element is copied from bit 1 of the corresponding value of \( q \). This instruction is unpredicated.

To compute \( \sin(x) \) or \( \cos(x) \) the instruction is executed with elements of the first source vector set to \( x \), adjusted to be in the range \(-\pi/4 < x \leq \pi/4\).

The elements of the second source vector hold the corresponding value of the quadrant \( q \) number as an integer not a floating-point value. The value \( q \) satisfies the relationship \((2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4\).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

Assembler Symbols

**<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.

**<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.

**<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSSel(element1, element2);
Z[d] = result;
```
INCB, INCD, INCH, INCW (scalar)

Increment scalar by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

Byte

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0 0 0 1 1 imm4 1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

size<1>size<0> D
```

```c
INCB <Xdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Doubleword

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0 1 1 1 imm4 1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

size<1>size<0> D
```

```c
INCD <Xdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Halfword

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0 1 1 imm4 1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

size<1>size<0> D
```

```c
INCH <Xdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

INCB, INCD, INCH, INCW (scalar)  Page 1949
INCW <Xdn>{, <pattern>{, MUL #<imm>}}</p>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;Xdn&gt;</th>
<th>Is the 64-bit name of the source and destination general-purpose register, encoded in the &quot;Rdn&quot; field.</th>
</tr>
</thead>
</table>
| <pattern> | Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101xx</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0xx</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x01x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xxx0</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1110x</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

| <imm> | Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field. |

### Operation

```plaintext
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(64) operand1 = X[dn];
X[dn] = operand1 + (count * imm);
```
INCD, INCH, INCW (vector)

Increment vector by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword, Halfword and Word

Doubleword

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | 1 | 1 | 1 | imm4 | 1 1 0 0 0 0 | pattern | Zdn
```

```plaintext
INCD <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Halfword

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | 0 | 1 | 1 | imm4 | 1 1 0 0 0 0 | pattern | Zdn
```

```plaintext
INCH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | 1 | 0 | 1 | imm4 | 1 1 0 0 0 0 | pattern | Zdn
```

```plaintext
INCW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx01</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx10</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + (count * imm);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INCP (scalar)

Increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 | size | 1 0 1 1 0 0 1 0 0 1 0 0 | Pm | Rdn |

INCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.
<T> Is the size specifier, encoded in “size”:

| size | <T> |
| 00 | B |
| 01 | H |
| 10 | S |
| 11 | D |

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bv(64) operand1 = X[dn];
bv(PL) operand2 = P[m];
integer count = 0;
for e = 0 to elements-1
    if ElemP(operand2, e, esize) == '1' then
        count = count + 1;
X[dn] = operand1 + count;
INCP (vector)

Increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
   if ElemP[operand2, e, esize] == '1' then
      count = count + 1;
for e = 0 to elements-1
   Elem[result, e, esize] = Elem[operand1, e, esize] + count;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INDEX (immediate, scalar)

Create index starting from immediate and incremented by general-purpose register

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

```
if !HaveSVE() then UNDEFINED;
integer size = 8 << UInt(size);
integer m = UInt(Rm);
integer d = UInt(Zd);
integer imm = SInt(imm5);
```

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:
  - size | <T>
  - 00 | B
  - 01 | H
  - 10 | S
  - 11 | D
- **<imm>** Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.
- **<R>** Is a width specifier, encoded in “size”:
  - size | <R>
  - 01 | W
  - x0 | W
  - 11 | X
- **<m>** Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(esize) operand2 = X[m];
integer element2 = SInt(operand2);
bits(VL) result;
for e = 0 to elements-1
    integer index = imm + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;
Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
INDEX (immediates)

Create index starting from and incremented by immediate

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element.

This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
</tr>
</tbody>
</table>

INDEX <Zd>.<T>, #<imm1>, #<imm2>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm1 = SInt(imm5);
integer imm2 = SInt(imm5b);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm1> Is the first signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<imm2> Is the second signed immediate operand, in the range -16 to 15, encoded in the "imm5b" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) result;
for e = 0 to elements-1
    integer index = imm1 + e * imm2;
    Elem[result, e, esize] = index<esize-1:0>;
Z[d] = result;
INDEX (scalar, immediate)

Create index starting from general-purpose register and incremented by immediate.

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

If !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
integer imm = SInt(imm5);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(esize) operand1 = X[n];
integer element1 = SInt(operand1);
bv(result);
for e = 0 to elements-1
    integer index = element1 + e * imm;
    Elem[result, e, esize] = index<esize-1:0>;
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
INDEX (scalars)

Create index starting from and incremented by general-purpose register

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Zd);
```

Assembler Symbols

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in “size”:
  - `size <T>`
    - 00: B
    - 01: H
    - 10: S
    - 11: D
- `<R>` is a width specifier, encoded in “size”:
  - `size <R>`
    - 01: W
    - x0: W
    - 11: X
- `<n>` is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- `<m>` is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(esize) operand1 = X[n];
integer element1 = SInt(operand1);
bits(esize) operand2 = X[m];
integer element2 = SInt(operand2);
bits(VL) result;
for e = 0 to elements-1
    integer index = element1 + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;
Z[d] = result;
```
INSR (scalar)

Insert general-purpose register in shifted vector

Shift the destination vector left by one element, and then place a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 0 0 0 0 0 1 0 1 size 1 0 0 1 0 0 0 0 1 1 1 0 | Rm | Zdn |

INSR <Zdn>.<T>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Rm);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operation

CheckSVEEnabled();
bits(VL) dest = Z[dn];
bits(esize) src = X[m];
Z[dn] = dest<(VL-esize)-1:0> : src;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INSR (SIMD&FP scalar)

Insert SIMD&FP scalar register in shifted vector

Shift the destination vector left by one element, and then place a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

INSR <Zdn>.<T>, <V><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Vm);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<m> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vm" field.

Operation

CheckSVEEnabled();
bits(VL) dest = Z[dn];
bits(esize) src = V[m];
Z[dn] = dest<VL-esize>-1:0> : src;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LASTA (scalar)

Extract element after last to general-purpose register

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then zero-extend and place the extracted element in the destination general-purpose register.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

LASTA <R><d>, <Pg>, <Zn>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = FALSE;

Assembler Symbols

| <R> | Is a width specifier, encoded in “size”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;R&gt;</td>
</tr>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;d&gt;</th>
<th>Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Pg&gt;</td>
<td>Is the name of the governing scalable predicate register P0-P7, encoded in the &quot;Pg&quot; field.</td>
</tr>
<tr>
<td>&lt;Zn&gt;</td>
<td>Is the name of the source scalable vector register, encoded in the &quot;Zn&quot; field.</td>
</tr>
</tbody>
</table>

| <T> | Is the size specifier, encoded in “size”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(Pg) mask = P[g];
bits(VL) operand = Z[n];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);
if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize]);
X[d] = result;
LASTA (SIMD&FP scalar)

Extract element after last to SIMD&FP scalar register

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then place the extracted element in the destination SIMD&FP scalar register.

LASTA <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = FALSE;

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
V[d] = Elem(operand, last, esize);
LASTB (scalar)

Extract last element to general-purpose register

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then zero-extend and place the extracted element in the destination general-purpose register.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| B | size | Pg | Zn | Rd |

LASTB <R><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = TRUE;

Assembler Symbols

<\textit{R}> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\textit{R}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<\textit{d}> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<\textit{Pg}> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<\textit{Zn}> Is the name of the source scalable vector register, encoded in the "Zn" field.

<\textit{T}> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\textit{T}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

\texttt{CheckSVEEnabled()};
integer elements = \texttt{VL} DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(rsize) result;
integer last = \texttt{LastActiveElement}(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
result = \texttt{ZeroExtend(Elem}(operand, last, esize));
X[d] = result;
LASTB (SIMD&FP scalar)

Extract last element to SIMD&FP scalar register

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then place the extracted element in the destination SIMD&FP register.

```
LASTB <V><d>, <Pg>, <Zn>.<T>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = TRUE;

**Assembler Symbols**

- **<V>** Is a width specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer last = LastActiveElement(mask, esize);
if isBefore then
  if last < 0 then last = elements - 1;
else
  last = last + 1;
if last >= elements then last = 0;
V[d] = Elem(operand, last, esize);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1B (scalar plus immediate)

Contiguous load unsigned bytes to vector (immediate index)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes:
- 8-bit element
- 16-bit element
- 32-bit element
- 64-bit element

8-bit element

```
1 0 1 0 0 0 1 0 0 0 0 0 0 imm4 1 0 1 Pg Rn Zt
```

dtype<3:1> dtype<0>

LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>:{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

16-bit element

```
1 0 1 0 0 0 1 0 0 0 0 0 0 imm4 1 0 1 Pg Rn Zt
```

dtype<3:1> dtype<0>

LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>:{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

```
1 0 1 0 0 0 1 0 0 0 0 0 0 imm4 1 0 1 Pg Rn Zt
```

dtype<3:1> dtype<0>

LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>:{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>]{, #<imm>, MUL VL}
LD1B (scalar plus scalar)

Contiguous load unsigned bytes to vector (scalar index)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1B {<Zt>.B}, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1B {<Zt>.H}, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1B (scalar plus scalar)
LD1B \{ <Zt> .S \}, <Pg>/Z, [<Xn|SP>, <Xm>]

if \! HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = Uint(Zt);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer g = Uint(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;

64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1B \{ <Zt> .D \}, <Pg>/Z, [<Xn|SP>, <Xm>]

if \! HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = Uint(Zt);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer g = Uint(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1B (scalar plus vector)

Gather load unsigned bytes to vector (vector index)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

32-bit unpacked unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \text{xs} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

LD1B \{ <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[
\begin{array}{cccccccccccccccccccccccc}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \text{xs} & 0 & \text{Zm} & 0 & 1 & 0 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

LD1B \{ <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & \text{Zm} & 1 & 1 & 0 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
**LD1B (vector plus immediate)**

Gather load unsigned bytes to vector (immediate index)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5| 1  | 1  | 0  | Pg | Zn | Zt |
```

LD1B \(\text{LD1B}\) \{<Zt>.S\}, \langle Pg/Z, [<Zn>.S\{, #<imm>\}\}]

if \(!\text{HaveSVE}()\) then UNDEFINED;

integer t = \text{UInt}(Zt);
integer n = \text{UInt}(Zn);
integer g = \text{UInt}(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = \text{UInt}(imm5);

### 64-bit element

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5| 1  | 1  | 0  | Pg | Zn | Zt |
```

LD1B \(\text{LD1B}\) \{<Zt>.D\}, \langle Pg/Z, [<Zn>.D\{, #<imm>\}\}]

if \(!\text{HaveSVE}()\) then UNDEFINED;

integer t = \text{UInt}(Zt);
integer n = \text{UInt}(Zn);
integer g = \text{UInt}(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = \text{UInt}(imm5);

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the “Zn” field.
- `<imm>` Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

\begin{verbatim}
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
\end{verbatim}

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1D (scalar plus immediate)

Contiguous load doublewords to vector (immediate index)

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 1 1 | 1 0 | imm4 | 1 0 | Pg | Rn | Zt
```

dtype<3:1>dtype<0>

LD1D { <Zt>.D }, <Pg>/Z, [ <Xn|SP>{, #<imm>, MUL VL}]

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

< Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```
LD1D (scalar plus scalar)

Contiguous load doublewords to vector (scalar index)

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

\[
\begin{array}{cccccccccccccccccccc}
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & Rm & 0 & 1 & 0 & Pg & Rn & Zt & \\
\end{array}
\]

dtype<3:1><dtype<0><

LD1D { <Zt>, D }, <Pg>/Z, [ <Xn|SP> ], <Xm>, LSL #3

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1D (scalar plus scalar)
LD1D (scalar plus vector)

Gather load doublewords to vector (vector index)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit unpacked scaled offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 1 1 xs 1 Zm 0 1 0 Pg Rn Zt
```

```
LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

32-bit unpacked unscaled offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 1 1 xs 0 Zm 0 1 0 Pg Rn Zt
```

```
LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit scaled offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 1 1 1 Zm 1 1 0 Pg Rn Zt
```

```
LD1D (scalar plus vector)
Page 1981
```
LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | Zm | 1  | 1  | 0  | Pg | Rn | Zt |

msz<1>msz<0> U ff

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

`CheckSVEEnabled();`
integer elements = \( \text{VL} \) DIV esize;
bits(64) base;
bits(PL) mask = \( \text{P}[g] \);
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then  
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then \( \text{SP}[\_] \) else \( \text{X}[n] \);
  offset = \( \text{Z}[m] \);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

\( \text{Z}[t] = \text{result}; \)
LD1D (vector plus immediate)

Gather load doublewords to vector (immediate index)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | imm5 | 1 | 1 | 0 | Pg | Zn | Zt |

LD1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

< Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1H (scalar plus immediate)

Contiguous load unsigned halfwords to vector (immediate index)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 0 1 0 0 1 0 | 0 1 0 | 1 | 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{{, #<imm>, MUL VL}}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 0 1 0 0 1 0 | 0 1 1 | 0 | 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{{, #<imm>, MUL VL}}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

64-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 1 0 1 0 0 1 0 | 0 1 1 | 1 | 0 | imm4 | 1 0 1 | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{{, #<imm>, MUL VL}}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1H (scalar plus scalar)

Contiguous load unsigned halfwords to vector (scalar index)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

LD1H { <Zt>.H }, <Pg>/Z, [Xn|SP>, Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

LD1H { <Zt>.S }, <Pg>/Z, [Xn|SP>, Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

LD1H (scalar plus scalar)
LD1H (scalar plus scalar)
LD1H (scalar plus vector)

Gather load unsigned halfwords to vector (vector index)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | xs| 1  | Zm| 0  | 1  | 0  | Pg | Rn | Zt |

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | xs| 1  | Zm| 0  | 1  | 0  | Pg | Rn | Zt |

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | xs| 0  | Zm| 0  | 1  | 0  | Pg | Rn | Zt |

msz<1>msz<0> U ff
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------|-----------------|
| 1 0 0 0 0 1 0 0 | Zm | 0 1 0 | Pg | Rn | Zt |

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------|-----------------|
| 1 1 0 0 0 1 0 0 | Zm | 1 1 0 | Pg | Rn | Zt |

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------|-----------------|
| 1 1 0 0 0 1 0 | Zm | 1 1 0 | Pg | Rn | Zt |

msz<1>msz<0> U ff
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrsoinUnpredictableBool(ConstrsoinUnpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
result = Z[t];
LD1H (vector plus immediate)

Gather load unsigned halfwords to vector (immediate index)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | Pg | Zn | Zt |

msz<1>msz<0> U ff

LD1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #imm}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | Pg | Zn | Zt |

msz<1>msz<0> U ff

LD1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #imm}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the “Zn” field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1RB

Load and broadcast unsigned byte to vector

Load a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate
offset which is in the range 0 to 63.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all
elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.
It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

LD1RB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

16-bit element

LD1RB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

32-bit element

LD1RB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
64-bit element

| 31302928272625 | 24 | 23 | 2221201918171615 | 14 | 13 | 121110 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------------|---|---|------------------|---|---|---------|---|---|---|---|---|---|---|---|---|---|---|
| 1 0 0 0 0 1 0 0 | 0 | 0 | 1 | imm6 | 1 | 1 | 1 | Pg | Rn | Zt |

dtypeh<1> dtypeh<0> 
dtypeh<1> dtypeh<0>

LD1RB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  bits(64) addr = base + offset * mbytes;
  data = Mem[addr, mbytes, AccType_SVE];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1RD

Load and broadcast doubleword to vector

Load a single doubleword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 8 in the range 0 to 504.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0   | 0   | 0   | 0   | 1   | 1   | 1   | imm6 | 1 | 1 | Pg | Rn | Zt | dtyphem<1> | dtyphem<0> | dtypel<1> | dtypel<0> |

LD1RD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1RH

Load and broadcast unsigned halfword to vector

Load a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

```
31302928272625 24 23 222120191817165 14 13 121110 9 8 7 6 5 4 3 2 1 0
| 1 0 0 0 1 0 | 0 | 1 | 1 | imm6 | 1 | 0 | 1 | Pg | Rn | Zt |
```

LD1RH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #imm}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

32-bit element

```
31302928272625 24 23 222120191817165 14 13 121110 9 8 7 6 5 4 3 2 1 0
| 1 0 0 0 1 0 | 0 | 1 | 1 | imm6 | 1 | 1 | 0 | Pg | Rn | Zt |
```

LD1RH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #imm}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

64-bit element

```
31302928272625 24 23 222120191817165 14 13 121110 9 8 7 6 5 4 3 2 1 0
| 1 0 0 0 1 0 | 0 | 1 | 1 | imm6 | 1 | 1 | 1 | Pg | Rn | Zt |
```

LD1RH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #imm}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
```
LD1ROB (scalar plus immediate)

Contiguous load and replicate thirty-two bytes (immediate index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE (FEAT_F64MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | imm4| 0  | 0  | 1  | Pg | Rn | Zt |

msz<1>msz<0> ssz

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]}

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.
Operation

`CheckSVEEnabled();`

if `VL` < 256 then UNDEFINED;

integer elements = 256 DIV `esize`;

bits(64) `base`;

bits(`PL`) `mask` = `P`[g]; // low bits only

bits(256) `result`;

constant integer `mbytes` = `esize` DIV 8;

if `HaveMTEExt()` then `SetTagCheckedInstruction`(n != 31);

if !`AnyActiveElement`(mask, `esize`) then
    if n == 31 && `ConstrainUnpredictableBool`(Unpredictable_CHECKSPNONEACTIVE) then
        `CheckSPAlignment`();
    else
        if n == 31 then `CheckSPAlignment`();
        base = if n == 31 then `SP`[] else `X`[n];

for e = 0 to `elements`-1
    if `ElemP`[mask, e, `esize`] == '1' then
        integer `eoff` = (offset * elements) + e;
        bits(64) `addr` = base + `eoff` * `mbytes`;
        `Elem`[result, e, `esize`] = `Mem`[`addr`, `mbytes`, `AccType_SVE`];
    else
        `Elem`[result, e, `esize`] = `Zeros`();

`Z`[t] = `ZeroExtend`(Replicate(result, `VL` DIV 256), `VL`);
LD1ROB (scalar plus scalar)

Contiguous load and replicate thirty-two bytes (scalar index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE
(FEAT_F64MM)

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```
LD1ROD (scalar plus immediate)

Contiguous load and replicate four doublewords (immediate index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored. ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE (FEAT_F64MM)

LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD1ROD (scalar plus scalar)**

Contiguous load and replicate four doublewords (scalar index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero. Only the first four predicate elements are used and higher numbered predicate elements are ignored. ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

**SVE (FEAT_F64MM)**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 0 1 0 | 1 1 0 1 | Rm | 0 0 0 | Pg | Rn | Zt |
```

msz<1>msz<0> ssz

**LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]**

```java
if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
```

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
if \( VL < 256 \) then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  ifElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
LD1ROH (scalar plus immediate)

Contiguous load and replicate sixteen halfwords (immediate index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE
(FEAT_F64MM)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[
\begin{array}{ccccccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \text{imm4} & 0 & 0 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

msz<1>msz<0> ssz

LD1ROH (scalar plus immediate)

\[
\text{LD1ROH} \{ <\text{Zt}>.H \}, <\text{Pg}>/Z, [<\text{Xn|SP}>{}, \#<\text{imm}>]}
\]

if \!HaveSVEFP64MatMulExt() then UNDEFINED;

integer \( t = \text{UInt}(\text{Zt}) \);
integer \( n = \text{UInt}(\text{Rn}) \);
integer \( g = \text{UInt}(\text{Pg}) \);
integer \( \text{esize} = 16 \);
integer \( \text{offset} = \text{SInt}(\text{imm4}) \);

Assembler Symbols

\(<\text{Zt}>\) Is the name of the scalable vector register to be transferred, encoded in the "\text{Zt}" field.

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "\text{Pg}" field.

\(<\text{Xn|SP}>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "\text{Rn}" field.

\(<\text{imm}>\) Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "\text{imm4}" field.
Operation

CheckSVEEnabled();
if \( VL < 256 \) then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = \( P[g] \); // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && Constr\( \text{ainUnpredictableBool}(\text{Unpredictable CHECKSPNONEACTIVE}) \) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then \( SP[] \) else \( X[n] \);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = ZeroExtend(Replicate(result, \( VL \) DIV 256), \( VL \));
**LD1ROH (scalar plus scalar)**

Contiguous load and replicate sixteen halfwords (scalar index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored. ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

**SVE (FEAT_F64MM)**

```
LD1ROH { <Zt>.H }, <Pg>/Z, [ <Xn|SP>, <Xm>, LSL #1 ]
```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
if $VL < 256$ then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits($PL$) mask = $P[g]$; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEEext() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then $SP[]$ else $X[n]$;
    offset = $X[m]$;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();

$Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);$
LD1ROW (scalar plus immediate)

Contiguous load and replicate eight words (immediate index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```plaintext
LD1ROW {<Zt>}.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
if \( VL < 256 \) then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();

\( Z[t] = \text{ZeroExtend(Replicate(result, VL DIV 256), VL);} \)
LD1ROW (scalar plus scalar)

Contiguous load and replicate eight words (scalar index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

SVE
(FEAT_F64MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>msz&lt;1&gt;</td>
<td>msz&lt;0&gt;</td>
<td>ssz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
if \( VL < 256 \) then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = \( P[g] \); // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else \( X[n] \);
    offset = \( X[m] \);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();

\( Z[t] \) = ZeroExtend(Replicate(result, VL DIV 256), VL);
**LD1RQB (scalar plus immediate)**

Contiguous load and replicate sixteen bytes (immediate index)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

- **Assembler Symbols**
  - `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
  - `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
  - `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
  - `<imm>` Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

- **Operation**

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (offset * 16) + (e * mbytes);
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = Replicate(result, VL DIV 128);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD1RQB (scalar plus scalar)**

Contiguous load and replicate sixteen bytes (scalar index)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Rm | Pg | Rn | Zt |

**Asmber Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

CheckSVEEnabled();

integer elements = 128 DIV esize;

bits(64) base;

bits(PL) mask = P[g]; // low 16 bits only

bits(64) offset;

bits(128) result;

constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if ! AnyActiveElement(mask, esize) then
if n == 31 then

ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then

CheckSPAlignment();

else

if n == 31 then

CheckSPAlignment();

base = if n == 31 then SP[] else X[n];

offset = X[m];

e = 0 to elements-1
if Elem[mask, e, esize] == '1' then

integer eoff = UINT(offset) + e;

bits(64) addr = base + eoff * mbytes;

Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];

else

Elem[result, e, esize] = Zeros();

Z[t] = Replicate(result, VL DIV 128);
LD1RQD (scalar plus immediate)

Contiguous load and replicate two doublewords (immediate index)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Active elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

inactive elements will not cause a read from Device memory or asignal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

Simple Elements Will Not Cause a Read from Device Memory or Signal a Fault, and Are Set to Zero. The Resulting Short Vector is Then Replicated to Fill the Long Destination Vector. Only the First Two Predicate Elements Are Used and Higher Numbered Predicate Elements Are Ignored.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

LD1RQD {<Zt>, <Pg>/Z, [<Xn|SP>{, #<imm}>]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (offset * 16) + (e * mbytes);
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = Replicate(result, VL DIV 128);
LD1RQD (scalar plus scalar)

Contiguous load and replicate two doublewords (scalar index)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 1 0 0 Rm 0 0 0 Pg 0 0 0 Zt
msz<1>msz<0> ssz

LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPA(fligment());
  else
    if n == 31 then CheckSPAignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = Replicate(result, VL DIV 128);
LD1RQH (scalar plus immediate)

Contiguous load and replicate eight halfwords (immediate index)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (offset * 16) + (e * mbytes);
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = Replicate(result, VL DIV 128);
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12, rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1RQH (scalar plus scalar)

Contiguous load and replicate eight halfwords (scalar index)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

![Assembler Symbols](image)

**Assembler Symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
        for e = 0 to elements-1
            if ElemP[mask, e, esize] == '1' then
                integer eoff = UInt(offset) + e;
                bits(64) addr = base + eoff * mbytes;
                Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
            else
                Elem[result, e, esize] = Zeros();
        Z[t] = Replicate(result, VL DIV 128);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
**LD1RQW (scalar plus immediate)**

Contiguous load and replicate four words (immediate index)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 0 1 0 0 1 0 | 1 | 0 | 0 | 0 | 0 | imm4 | 0 | 0 | 1 |

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

**Operation**

```pseudocode
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
```

If !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);

integer n = UInt(Rn);

integer g = UInt(Pg);

integer esize = 32;

integer offset = SInt(imm4);

```pseudocode
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
```

If !HaveMTEExt() then SetTagCheckedInstruction(n != 31);

```pseudocode
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            bits(64) addr = base + (offset * 16) + (e * mbytes);
            Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[result, e, esize] = Zeros();
    Z[t] = Replicate(result, VL DIV 128);
```

If !AnyActiveElement(mask, esize) then

if !AnyActiveElement(mask, esize) then

if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then

    CheckSPAlignment();

else

    if n == 31 then CheckSPAlignment();

    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1

    if ElemP[mask, e, esize] == '1' then

        bits(64) addr = base + (offset * 16) + (e * mbytes);

        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];

    else

        Elem[result, e, esize] = Zeros();

    Z[t] = Replicate(result, VL DIV 128);
LD1RQW (scalar plus scalar)

Contiguous load and replicate four words (scalar index)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

 active elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  |

msz<1>msz<0> ssz

LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '1111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
  for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + e;
      bits(64) addr = base + eoff * mbytes;
      Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[result, e, esize] = Zeros();
  Z[t] = Replicate(result, VL DIV 128);
LD1RSB

Load and broadcast signed byte to vector

Load a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element.

16-bit element

LD1RSB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

32-bit element

LD1RSB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

64-bit element

LD1RSB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  bits(64) addr = base + offset * mbytes;
  data = Mem[addr, mbytes, AccType_SVE];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1RSH

Load and broadcast signed halfword to vector

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

31302928272625 24 23 2221201918171615 14 13 121110 9 8 7 6 5 4 3 2 1 0

| 1 0 0 0 1 0 | 1 | 0 | 1 | imm6 | 1 | 0 | 1 | Pg | Rn | Zt |

If !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

64-bit element

31302928272625 24 23 2221201918171615 14 13 121110 9 8 7 6 5 4 3 2 1 0

| 1 0 0 0 1 0 | 1 | 0 | 1 | imm6 | 1 | 0 | 0 | Pg | Rn | Zt |

If !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1RSW

Load and broadcast signed word to vector

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

```
LD1RSW {
  <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

Operation

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

```
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  bits(64) addr = base + offset * mbytes;
  data = Mem[addr, mbytes, AccType_SVE];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```
**LD1RW**

Load and broadcast unsigned word to vector

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
31302928272625  24  23  2221201918171615  14  13  121110  9  8  7  6  5  4  3  2  1  0
1 0 0 0 1 0  1 0 1  1  1  1  0  Pg  Rn  Zt
```

LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

### 64-bit element

```
31302928272625  24  23  2221201918171615  14  13  121110  9  8  7  6  5  4  3  2  1  0
1 0 0 0 1 0  1 0 1  1  1  1  1  Pg  Rn  Zt
```

LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        bits(64) addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1SB (scalar plus immediate)

Contiguous load signed bytes to vector (immediate index)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
integer n = UINT(Rn);
integer g = UINT(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SINT(imm4);

32-bit element

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
integer n = UINT(Rn);
integer g = UINT(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SINT(imm4);

64-bit element

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
integer n = UINT(Rn);
integer g = UINT(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SINT(imm4);
Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD1SB (scalar plus scalar)**

Contiguous load signed bytes to vector (scalar index)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: **16-bit element**, **32-bit element** and **64-bit element**

### 16-bit element

```
16-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 1 1 0 0 | Rm | 0 1 0 | Pg | Rn | Zt

dtype<3:1>dtype<0>
```

**LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]**

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;

### 32-bit element

```
32-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 1 0 1 1 | Rm | 0 1 0 | Pg | Rn | Zt

dtype<3:1>dtype<0>
```

**LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]**

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;

### 64-bit element

```
64-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 1 0 0 0 | Rm | 0 1 0 | Pg | Rn | Zt

dtype<3:1>dtype<0>
```
LD1SB \( <\text{Zt}>.D \), \(<\text{Pg}>/Z, [<\text{Xn}|\text{SP}>, <\text{Xm}>] \)

if !\texttt{HaveSVE}() then UNDEFINED;
if \( Rm == '11111' \) then UNDEFINED;
integer \( t = \texttt{UInt}(\text{Zt}) \);
integer \( n = \texttt{UInt}(\text{Rn}) \);
integer \( m = \texttt{UInt}(\text{Rm}) \);
integer \( g = \texttt{UInt}(\text{Pg}) \);
integer \( \text{esize} = 64 \);
integer \( \text{msize} = 8 \);
boolean \( \text{unsigned} = \text{FALSE} \);

Assembler Symbols

\texttt{<Zt>} \quad \text{Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.}
\texttt{<Pg>} \quad \text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.}
\texttt{<Xn|SP>} \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}
\texttt{<Xm>} \quad \text{Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.}

Operation

\texttt{CheckSVEEnabled}();
integer \( \text{elements} = \texttt{VL} \div \text{esize} \);
bits(64) \( \text{base} \);
bits(\text{PL}) \( \text{mask} = P[g] \);
bits(\text{VL}) \( \text{result} \);
bits(\text{msize}) \( \text{data} \);
bits(64) \( \text{offset} \);
constant integer \( \text{mbytes} = \text{msize} \div 8 \);
if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(\text{TRUE});
if !\texttt{AnyActiveElement}(\text{mask}, \text{esize}) then
  if \( n == 31 \) \&\& \texttt{ConstrainUnpredictableBool}(\texttt{Unpredictable\_CHECKSPNONEACTIVE}) then
    \texttt{CheckSPAlignment}();
  else
    if \( n == 31 \) then \texttt{CheckSPAlignment}();
    \( \text{base} = \text{if} \ n == 31 \ \text{then} \ \texttt{SP}[] \ \text{else} \ \texttt{X}[n] \);
    \( \text{offset} = \texttt{X}[m] \);
for \( e = 0 \) to \text{elements}-1
  if \texttt{ElemP}(\text{mask}, e, \text{esize}) == '1'
    \( \text{bits}(64) \ \text{addr} = \text{base} + (\texttt{UInt}(\text{offset}) + e) * \text{mbytes} \);
    \( \text{data} = \texttt{Mem}[\text{addr}, \text{mbytes}, \texttt{AccType\_SVE}] \);
    \( \text{Elem}[\text{result}, e, \text{esize}] = \texttt{Extend}(\text{data}, \text{esize}, \text{unsigned}) \);
  else
    \( \text{Elem}[\text{result}, e, \text{esize}] = \texttt{Zeros}() \);
\( Z[t] = \text{result} \);
LD1SB (scalar plus vector)

Gather load signed bytes to vector (vector index)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

LD1SB (scalar plus vector)
LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    else base = if n == 31 then SP[] else X[n];
    offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[off, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  Z[t] = result;
LD1SB (vector plus immediate)

Gather load signed bytes to vector (immediate index)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  |
| msz<1> | msz<0> | U | ff |

ld1sb { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  |
| msz<1> | msz<0> | U | ff |

ld1sb { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1SH (scalar plus immediate)

Contiguous load signed halfwords to vector (immediate index)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bites(64) base;
bite(PL) mask = P[g];
bite(VL) result;
bite(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
bite(64) addr = base + eoff * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1SH (scalar plus scalar)

Contiguous load signed halfwords to vector (scalar index)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;

64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEEext() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
```
**LD1SH (scalar plus vector)**

Gather load signed halfwords to vector (vector index)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: **32-bit scaled offset**, **32-bit unpacked scaled offset**, **32-bit unpacked unscaled offset**, **32-bit unscaled offset**, **64-bit scaled offset** and **64-bit unscaled offset**

### 32-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 1  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 1  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | xs | 0  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

msz<1>msz<0>
LD1SH \{ <Zt>.D \}, \langle Pg/Z, [<Xn|SP>, <Zm>.D,<mod>]\}

if \! \texttt{HaveSVE()} \text{ then UNDEFINED;}
integer t = \texttt{UInt}(Zt);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Zm);
integer g = \texttt{UInt}(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

\[
\begin{array}{ccccccccccccccccccccccc}
\hline
1 & 0 & 0 & 0 & 1 & 0 & 1 & xs & 0 & Zm & 0 & 0 & 0 & Pg & Rn & Zt \\
\end{array}
\]

LD1SH \{ <Zt>.S \}, \langle Pg/Z, [<Xn|SP>, <Zm>.S,<mod]\}

if \! \texttt{HaveSVE()} \text{ then UNDEFINED;}
integer t = \texttt{UInt}(Zt);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Zm);
integer g = \texttt{UInt}(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
LD1SH \{<Zt>.D\}, \langle \text{Pg}/\text{Z}, [<Xn|SP>, <Zm>.D]\}

if !\text{HaveSVE}() then UNDEFINED;
integer t = \text{UInt}(<Zt>);
integer n = \text{UInt}(<Rn>);
integer m = \text{UInt}(<Zm>);
integer g = \text{UInt}(<Pg>);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL \text{ DIV esize};
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize \text{ DIV 8};
if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(TRUE);
if !\text{AnyActiveElement}(mask, esize) then
  if n == 31 && \text{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) then
    \text{CheckSPAlignment}();
  else
    if n == 31 then \text{CheckSPAlignment}();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
  for e = 0 to elements-1
    if \text{ElemP}[mask, e, esize] == '1' then
      integer off = \text{Int}(\text{Elem}[offset, e, esize]<offs_size-1:0>, offs_unsigned);
      bits(64) addr = base + (off << scale);
      data = \text{Mem}[addr, mbytes, \text{AccType SVE}];
      \text{Elem}[result, e, esize] = \text{Extend}(data, esize, unsigned);
    else
      \text{Elem}[result, e, esize] = \text{Zeros}();
  \text{Z}[t] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1SH (vector plus immediate)

Gather load signed halfwords to vector (immediate index)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**LD1SH** { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**LD1SH** { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

**Assembler Symbols**

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
- **<Zn>** Is the name of the base scalable vector register, encoded in the “Zn” field.
- **<imm>** Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1SW (scalar plus immediate)

Contiguous load signed words to vector (immediate index)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector’s in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 1 0 0 1 0 0 1 0 0 0imm4 1 0 1 Pg Rn Zt
```

dtype<3:1>dtype<0>

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then
      CheckSPAlignment();
     base = if n == 31 then SP[] else X[n];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1'
      integer eoff = (offset * elements) + e;
      bits(64) addr = base + eoff * mbytes;
      data = Mem[addr, mbytes, AccType_SVE];
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
      Elem[result, e, esize] = Zeros();
  Z[t] = result;
LD1SW (scalar plus scalar)

Contiguous load signed words to vector (scalar index)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
1 0 1 0 0 1 0 | 0 1 0 | 0  | Rm | 0 1 0 | Pg | Rn | Zt |
```

dtype<3:1>dtype<0>

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
    end if
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
    end if
end for
Z[t] = result;
```
LD1SW (scalar plus vector)

Gather load signed words to vector (vector index)

Gather load of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes:
- 32-bit unpacked scaled offset
- 32-bit unpacked unscaled offset
- 64-bit scaled offset
- 64-bit unscaled offset

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 1  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

LD1SW {<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

### 32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 0  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

LD1SW {<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | Zm | 1  | 0  | 0  | Pg | Rn | Zt |

LD1SW (scalar plus vector)
LD1SW \( \{ <Zt>.D \} \), \( <Pg>/Z \), \( [<Xn|SP>, <Zm>.D, LSL \#2] \)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer \( t = \text{UInt}(Zt) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Zm) \);
integer \( g = \text{UInt}(Pg) \);
integer \( \text{esize} = 64 \);
integer \( \text{msize} = 32 \);
integer \( \text{offs\_size} = 64 \);
boolean \( \text{unsigned} = \text{FALSE} \);
boolean \( \text{offs\_unsigned} = \text{TRUE} \);
integer \( \text{scale} = 2 \);

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

LD1SW \( \{ <Zt>.D \} \), \( <Pg>/Z \), \( [<Xn|SP>, <Zm>.D] \)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer \( t = \text{UInt}(Zt) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Zm) \);
integer \( g = \text{UInt}(Pg) \);
integer \( \text{esize} = 64 \);
integer \( \text{msize} = 32 \);
integer \( \text{offs\_size} = 64 \);
boolean \( \text{unsigned} = \text{FALSE} \);
boolean \( \text{offs\_unsigned} = \text{TRUE} \);
integer \( \text{scale} = 0 \);

Assembler Symbols

\(<Zt>\)  
Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

\(<Pg>\)  
Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Xn|SP>\)  
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<Zm>\)  
Is the name of the offset scalable vector register, encoded in the "Zm" field.

\(<\text{mod}>\)  
Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>(&lt;\text{mod}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = Z[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1SW (vector plus immediate)

Gather load signed words to vector (immediate index)

Gather load of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

LD1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP(mask, e, esize) == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
data = Mem[addr, mbytes, AccType_SVE];
Elem[result, e, esize] = Extend(data, esize, unsigned);
else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1W (scalar plus immediate)

Contiguous load unsigned words to vector (immediate index)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 0 1 | 0 | 0 | imm4 | 1 0 1 | Pg | Rn | Zt

dtype<3:1> dtype<0>

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

64-bit element

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 0 1 | 1 | 0 | imm4 | 1 0 1 | Pg | Rn | Zt

dtype<3:1> dtype<0>

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1W (scalar plus scalar)

Contiguous load unsigned words to vector (scalar index)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|-------------|----------------|
| 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Rm 0 1 0 | Pg 10 | Rn 11 | Zt 12 |

LD1W ( <Zt>.S }, <Pg>/Z, [ <Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;

64-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|-------------|----------------|
| 0 1 0 1 0 1 1 1 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Rm 0 1 0 | Pg 10 | Rn 11 | Zt 12 |

LD1W ( <Zt>.D }, <Pg>/Z, [ <Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`
integer elements = `VL` DIV `esize`;
bits(64) base;
bits(`PL`) mask = `P`[g];
bits(`VL`) result;
bits(`msize`) data;
bits(64) offset;
constant integer mbytes = `msize` DIV 8;

if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`

if !`AnyActiveElement`(mask, `esize`) then
    if n == 31 && `ConstrainUnpredictableBool`(Unpredictable_CHECKSPNONEACTIVE) then
        `CheckSPAlignment();`
    else
        if n == 31 then `CheckSPAlignment();`
        base = if n == 31 then `SP`[] else `X`[n];
        offset = `X`[m];

for e = 0 to elements-1
    if `ElemP`(mask, e, `esize`) == '1' then
        bits(64) addr = base + (`UInt`(offset) + e) * mbytes;
        data = `Mem`[addr, mbytes, AccType_SVE];
        `Elem`[result, e, `esize`] = `Extend`(data, `esize`, unsigned);
    else
        `Elem`[result, e, `esize`] = `Zeros`();

`Z`[t] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1W (scalar plus vector)

Gather load unsigned words to vector (vector index)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 1  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 1  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 0  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |

msz<1>msz<0>
LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>xs</td>
<td>0</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1W (scalar plus vector)
LD1W \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
iinteger elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 \&\& ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
      bits(64) addr = base + (off << scale);
      data = Mem[addr, mbytes, AccType_SVE];
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
      Elem[result, e, esize] = Zeros();
  Z[t] = result;

**LD1W (vector plus immediate)**

Gather load unsigned words to vector (immediate index)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>1 0</td>
<td>0 1</td>
<td>imm5</td>
</tr>
</tbody>
</table>

**LD1W** \( <\text{Zt}.S}, <\text{Pg}/Z, [<\text{Zn}.S{, #}<\text{imm}>} \)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>1 0</td>
<td>0 1</td>
<td>imm5</td>
</tr>
</tbody>
</table>

**LD1W** \( <\text{Zt}.D}, <\text{Pg}/Z, [<\text{Zn}.D{, #}<\text{imm}>} \)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

**Assembler Symbols**

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
- **<Zn>** Is the name of the base scalable vector register, encoded in the “Zn” field.
- **<imm>** Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.
Operation

`CheckSVEEnabled();`
`integer elements = VL DIV esize;`
`bits(PL) mask = P[g];`
`bits(VL) base;`
`bits(VL) result;`
`bits(msize) data;`
`constant integer mbytes = msize DIV 8;`
`if HaveMTEExt() then SetTagCheckedInstruction(TRUE);`
`if AnyActiveElement(mask, esize) then
    base = Z[n];`
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem(addr, mbytes, AccType_SVE);
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;`
LD2B (scalar plus immediate)

Contiguous load two-byte structures to two vectors (immediate index)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
LD2B (scalar plus scalar)

Contiguous load two-byte structures to two vectors (scalar index)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | Rm | 1  | 1  | 0  | Pg | 1  | Rn | 1  | Zt |
```

LD2B \{ <Zt1>.B, <Zt2>.B \}, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 2;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bite(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

LD2D (scalar plus immediate)

Contiguous load two-doubleword structures to two vectors (immediate index)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

```assembly
LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 2;
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTExe() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
bits(64) addr = base + eoff * mbytes;
Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
else
    Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
LD2D (scalar plus scalar)

Contiguous load two-doubleword structures to two vectors (scalar index)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------|
| 1 0 1 0 0 1 0 1 1 0 1           | Rm 1 1 0                         | Pg      |
|                                  | Rn Zt                            |         |
| msz<1>msz<0>                    |                                  |         |

LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD2H (scalar plus immediate)**

Contiguous load two-halfword structures to two vectors (immediate index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
LD2H (scalar plus immediate)

Contiguous load two-halfword structures to two vectors (immediate index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

    31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    1 0 1 0 0 1 0 | 0 1 0 1 0 | imm4 1 1 1 | Pg | Rn | Zt

    msz<1> msz<0>

LD2H {<Zt1>.H, <Zt2>.H}, < Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
iconteger n = UInt(Rn);
iconteger g = UInt(Pg);
iconteger esize = 16;
iconteger offset = SInt(imm4);
iconteger nreg = 2;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v33.16decrel,AdvSIMD v29.05,pseudocode v2021-12_rel,sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD2H (scalar plus scalar)**

Contiguous load two-halfword structures to two vectors (scalar index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 16 15</th>
<th>14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
<td>0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>Rm</td>
<td>Pg</td>
<td>Rn</td>
</tr>
<tr>
<td>Zt</td>
<td></td>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

**LD2H** { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 2;
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`

integer elements = \textit{VL} DIV esize;

bits(64) base;

bits(PL) mask = \textit{P}[g];

bits(64) offset;

constant integer mbytes = esize DIV 8;

array [0..1] of bits(\textit{VL}) values;

if \textit{HaveMTEExt}() then \textit{SetTagCheckedInstruction}(TRUE);

if \texttt{AnyActiveElement}(mask, esize) then

  if \texttt{n == 31} \&\& \textit{ConstrainUnpredictableBool}(\textit{Unpredictable_CHECKSPNONEACTIVE}) then
  \hspace*{1em} \textit{CheckSPAlignment}();

else

  if \texttt{n == 31} then \textit{CheckSPAlignment}();

  base = if \texttt{n == 31} then \textit{SP}[] else \textit{X}[n];

  offset = \textit{X}[m];

for e = 0 to elements-1

  for r = 0 to nreg-1

    if \texttt{ElemP[mask, e, esize]} == '1' then

      integer eoff = \texttt{UInt}(offset) + (e * nreg) + r;

      bits(64) addr = base + eoff * mbytes;

      \textit{Elem}[values[r], e, esize] = \textit{Mem}[addr, mbytes, \textit{AccType_SVE}];

    else

      \textit{Elem}[values[r], e, esize] = \texttt{Zeros}();

  for r = 0 to nreg-1

    \texttt{Z}[(t+r) MOD 32] = values[r];
LD2W (scalar plus immediate)

Contiguous load two-word structures to two vectors (immediate index)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
ninteger n = UInt(Rn);
ginteger g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
ninteger nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \text{ DIV} \texttt{esize};

bits(64) base;
bits(PL) mask = \texttt{P}[g];

constant integer mbytes = \texttt{esize} \text{ DIV} 8;

array [0..1] of bits(\texttt{VL}) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 \&\& constrainUnpredictableBool(\texttt{Unpredictable_CHECKSPNONEACTIVE}) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then \texttt{SP}[] else \texttt{X}[n];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if \texttt{ElemP[mask, e, esize]} == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      \texttt{Elem[values[r], e, esize]} = \texttt{Mem}[addr, mbytes, \texttt{AccType_SVE}];
    else
      \texttt{Elem[values[r], e, esize]} = \texttt{Zeros}();

for r = 0 to nreg-1
  \texttt{Z}[(t+r) \text{ MOD} 32] = values[r];
LD2W (scalar plus scalar)

Contiguous load two-word structures to two vectors (scalar index)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | Pg | Rn | Zt |

msz<1>msz<0>

LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;  
bits(PL) mask = P[g];
bits(64) offset;  
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then  
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then  
        CheckSPAlignment();
    else
        if n == 31 then  
            base = if n == 31 then SP[] else X[n];
            offset = X[m];
        for e = 0 to elements-1
            for r = 0 to nreg-1
                if ElemP[mask, e, esize] == '1' then
                    integer eoff = UInt(offset) + (e * nreg) + r;
                    bits(64) addr = base + eoff * mbytes;
                    Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
                else
                    Elem[values[r], e, esize] = Zeros();
            for r = 0 to nreg-1
                Z[(t+r) MOD 32] = values[r];
```
LD3B (scalar plus immediate)

Contiguous load three-byte structures to three vectors (immediate index)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD3B (scalar plus scalar)

Contiguous load three-byte structures to three vectors (scalar index)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
msz<1>msz<0>

LD3B \{ <Zt1>.B, <Zt2>.B, <Zt3>.B \}, <Pg>/Z, [<Xn|SP>, < Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD3D (scalar plus immediate)

Contiguous load three-doubleword structures to three vectors (immediate index)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 0 1 0 0 1 0 | 1 | 1 | 1 0 | 0 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |

LD3D \{ <Zt1>.D, <Zt2>.D, <Zt3>.D \}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD3D (scalar plus scalar)

Contiguous load three-doubleword structures to three vectors (scalar index)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
LD3D {<Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

If `!HaveSVE()` then UNDEFINED;
If `Rm == '11111'` then UNDEFINED;
Integer `t = UInt(Zt);`
Integer `n = UInt(Rn);`
Integer `m = UInt(Rm);`
Integer `g = UInt(Pg);`
Integer `esize = 64;`
Integer `nreg = 3;`

**Assembler Symbols**

- `<Zt1>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros();
    for r = 0 to nreg-1
        Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD3H (scalar plus immediate)

Contiguous load three-halfword structures to three vectors (immediate index)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD3H (scalar plus scalar)

Contiguous load three-halfword structures to three vectors (scalar index)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [ <Xn|SP>, <Xm>, LSL #1 ]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 3;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD3W (scalar plus immediate)

Contiguous load three-word structures to three vectors (immediate index)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 3;
Operation

CheckSVEEnabled();
integer elements = VL \text{ DIV} esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize \text{ DIV} 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset \times elements \times nreg) + (e \times nreg) + r;
bits(64) addr = base + eoff \times mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) \text{ MOD} 32] = values[r];
**LD3W (scalar plus scalar)**

Contiguous load three-word structures to three vectors (scalar index)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

![Memory Access Diagram](image)

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```assembly
LD3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
in integer t = UINT(Zt);
in integer n = UINT(Rn);
in integer m = UINT(Rm);
in integer g = UINT(Pg);
in integer esize = 32;
in integer nreg = 3;

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD4B (scalar plus immediate)

Contiguous load four-byte structures to four vectors (immediate index)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

![Binary Representation of LD4B Instruction](image)


if !HaveSVE() then UNDEFINED;
in\text{teger} \ t = \text{UInt}(Zt);
in\text{teger} \ n = \text{UInt}(Rn);
in\text{teger} \ g = \text{UInt}(Pg);
in\text{teger} \ esize = 8;
in\text{teger} \ offset = \text{SInt}(imm4);
in\text{teger} \ nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
Pg Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
     if n == 31 && constrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
       CheckSPAlignment();
else
     if n == 31 then CheckSPAlignment();
     base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
   for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
         integer eoff = (offset * elements * nreg) + (e * nreg) + r;
         bits(64) addr = base + eoff * mbytes;
         Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
      else
         Elem[values[r], e, esize] = Zeros();
   for r = 0 to nreg-1
      Z[ (t+r) MOD 32 ] = values[r];
LD4B (scalar plus scalar)

Contiguous load four-byte structures to four vectors (scalar index)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

![Binary representation of LD4B instruction](image)

**Assembler Symbols**

- `<Zt1>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```assembly
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 4;
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1'
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD4D (scalar plus immediate)

Contiguous load four-doubleword structures to four vectors (immediate index)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 1 1 0 imm4 1 1 1 Pg Rn Zt


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```
LD4D (scalar plus scalar)

Contiguous load four-doubleword structures to four vectors (scalar index)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```
LD4H (scalar plus immediate)

Contiguous load four-halfword structures to four vectors (immediate index)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure.Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

 inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.


Assembler Symbols

- $<Zt1>$ Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- $<Zt2>$ Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- $<Zt3>$ Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- $<Zt4>$ Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- $<Pg>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- $<Xn|SP>$ Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- $<\text{imm}>$ Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD4H (scalar plus scalar)

Contiguous load four-halfword structures to four vectors (scalar index)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

- `CheckSVEEnabled();`
- `integer elements = VL DIV esize;`
- `bits(64) base;`
- `bits(PL) mask = P[g];`
- `bits(64) offset;`
- `constant integer mbytes = esize DIV 8;`
- `array [0..3] of bits(VL) values;`

If `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`

If `!AnyActiveElement(mask, esize)` then
  - If `n == 31` and `ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)` then `CheckSPAlignment();`
  - Else
    - If `n == 31` then `CheckSPAlignment();`
    - `base = if n == 31 then SP[] else X[n];`
    - `offset = X[m];`

For `e = 0 to elements-1`
  - For `r = 0 to nreg-1`
    - If `ElemP[mask, e, esize] == '1'` then
      - `integer eoff = UInt(offset) + (e * nreg) + r;`
      - `bits(64) addr = base + eoff * mbytes;`
      - `Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];`
    - Else
      - `Elem[values[r], e, esize] = Zeros();`

For `r = 0 to nreg-1`
  - `Z[(t+r) MOD 32] = values[r];`
LD4W (scalar plus immediate)

Contiguous load four-word structures to four vectors (immediate index)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address
generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied
by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four
consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device
memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros();
    for r = 0 to nreg-1
        Z[(t+r) MOD 32] = values[r];
**LD4W (scalar plus scalar)**

Contiguous load four-word structures to four vectors (scalar index)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 4;

**Assembler Symbols**

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
      Elem[values[r], e, esize] = Zeros();
  for r = 0 to nreg-1
    Zf[(t+r) MOD 32] = values[r];
LDFF1B (scalar plus scalar)

Contiguous load first-fault unsigned bytes to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

```
LDFF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
```

16-bit element

```
LDFF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
```

32-bit element

```
LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
broadcast base;
broadcast(PL) mask = P[g];
broadcast(VL) result;
broadcast(VL) orig = Z[t];
broadcast(msize) data;
broadcast(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    end
    (data, fault) = (Zeros(msize), FALSE);
  end
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
    if unknown then
      if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros();
      else
        Elem[result, e, esize] = Elem[orig, e, esize];
      end
    end
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  end
Z[t] = result;
LDFF1B (scalar plus vector)

Gather load first-fault unsigned bytes to vector (vector index)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset

32-bit unpacked unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|----------------|
| 1 1 0 0 0 1 0 | 0 0 xs 0 | Zm 0 1 1 | Pg | Rn | Zt |
| msz<1>msz<0> |

LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|----------------|
| 1 0 0 0 1 0 | 0 0 xs 0 | Zm 0 1 1 | Pg | Rn | Zt |

LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|----------------|
| 1 1 0 0 0 1 0 | 0 0 1 0 | Zm 1 1 1 | Pg | Rn | Zt |
| msz<1>msz<0> |

LDFF1B (scalar plus vector)
LDFF1B \{ <Zt>, <Pg>/Z, [<Xn|SP>, <Zm>] \}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = Z[m];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
            bits(64) addr = base + (off << scale);
            if first then
                // Mem[] will not return if a fault is detected for the first active element
                data = Mem[addr, mbytes, AccType_SVE];
                first = FALSE;
            else
                // MemNF[] will return fault=TRUE if access is not performed for any reason
                (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
            else
                (data, fault) = (Zeros(msize), FALSE);
            // FFR elements set to FALSE following a supressed access/fault
            faulted = faulted || fault;
            if faulted then
                ElemFFR[e, esize] = '0';
            // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
            unknown = unknown || ElemFFR[e, esize] == '0';
            if unknown then
                if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                    Elem[result, e, esize] = Extend(data, esize, unsigned);
                elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                    Elem[result, e, esize] = Zeros();
                else // merge
                    Elem[result, e, esize] = Elem[orig, e, esize];
                else
                    Elem[result, e, esize] = Extend(data, esize, unsigned);
            Z[t] = result;
LDFF1B (vector plus immediate)

Gather load first-fault unsigned bytes to vector (immediate index)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LDFF1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LDFF1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
            else
                Elem[result, e, esize] = Extend(data, esize, unsigned);
        Z[t] = result;
```
LDFF1D (scalar plus scalar)

Contiguous load first-fault doublewords to vector (scalar index)

Contiguous load with first-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

LDFF1D \{ <Zt>.D, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  if Elemp[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElempFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t] = result;
**LDFF1D (scalar plus vector)**

Gather load first-fault doublewords to vector (vector index)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | xs| 1  | Zm | 0  | 1  | 1  | Pg | Rn | Zt |

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;

### 32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | xs| 0  | Zm | 0  | 1  | 1  | Pg | Rn | Zt |

msz<1>msz<0> U ff

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1 | Zm | 1 | 1 | 1 | Pg | Rn | Zt |

U ff
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

### 64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  |

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

### Assembler Symbols

- **<Zt>**
  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

- **<Pg>**
  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- **<Xn|SP>**
  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- **<Zm>**
  Is the name of the offset scalable vector register, encoded in the "Zm" field.

- **<mod>**
  Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];

for e = 0 to elements-1
  if Elemp[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize], offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);

// FFR elements set to FALSE following a supressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros();
  else
    Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1D (vector plus immediate)

Gather load first-fault doublewords to vector (immediate index)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

\[
\begin{array}{ccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & \text{imm5} & 1 & 1 & 1 & \text{Pg} & \text{Zn} & \text{Zt} \\
\end{array}
\]

\[
\text{msz}<1>\text{msz}<0> \quad \text{U ff}
\]

\[
\text{LDFF1D} \{ \text{<Zt>.D} \}, \text{<Pg>/Z}, \text{[<Zn>.D{}, #<imm>]} \}
\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer \(t = \text{UInt}(\text{Zt});\)
integer \(n = \text{UInt}(\text{Zn});\)
integer \(g = \text{UInt}(\text{Pg});\)
integer \(\text{esize} = 64;\)
integer \(\text{msize} = 64;\)
boolean \(\text{unsigned} = \text{TRUE};\)
integer \(\text{offset} = \text{UInt}(\text{imm5});\)

Assembler Symbols

\(<\text{Zt}>\)
Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

\(<\text{Pg}>\)
Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zn}>\)
Is the name of the base scalable vector register, encoded in the "Zn" field.

\(<\text{imm}>\)
Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits[PL] mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        end
    end
    (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else
                Elem[result, e, esize] = Elem[orig, e, esize];
            end
        end
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    end
Z[t] = result;
**LDFF1H (scalar plus scalar)**

Contiguous load first-fault unsigned halfwords to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

### 16-bit element

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|----------------|----------------|----------------|
| 1 0 1 0 0 1 0 | 0 1 0 1 1       | Rm             | 0 1 1          | Pg             | Rn             | Zt             |

dtype<3:1>dtype<0>
```

```
LDFF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm}, LSL #1]}
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
```

### 32-bit element

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|----------------|----------------|----------------|
| 1 0 1 0 0 1 0 | 0 1 1 0         | Rm             | 0 1 1          | Pg             | Rn             | Zt             |

dtype<3:1>dtype<0>
```

```
LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm}, LSL #1]}
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
```

### 64-bit element

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|----------------|----------------|----------------|
| 1 0 1 0 0 1 0 | 0 1 1 1         | Rm             | 0 1 1          | Pg             | Rn             | Zt             |

dtype<3:1>dtype<0>
```

```
LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm}, LSL #1]}
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
```
### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 & & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRANDED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
```

LDFF1H (scalar plus scalar)
LDFF1H (scalar plus vector)

Gather load first-fault unsigned halfwords to vector (vector index)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit scaled offset

| 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 1  | Zm | 0  | 1  | 1  | Pg | Rn | Zt |

LDFF1H ( <Zt> .S ), <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked scaled offset

| 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 1  | 0  | 0  | 1  | 0  | 1  | xs | 1  | Zm | 0  | 1  | 1  | Pg | Rn | Zt |

LDFF1H ( <Zt>.D ), <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked unscaled offset

| 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | xs | 0  | Zm | 0  | 1  | 1  | Pg | Rn | Zt |

LDFF1H (scalar plus vector)
LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>U ff</td>
</tr>
</tbody>
</table>

LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>U ff</td>
</tr>
</tbody>
</table>

LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt; U ff</td>
</tr>
</tbody>
</table>
LDFF1H (scalar plus vector)
Operation

`CheckSVEEnabled();`
`integer elements = VL DIV esize;`
`bits(PL) mask = P[g];`
`bits(64) base;`
`bits(VL) offset;`
`bits(VL) result;`
`bits(VL) orig = Z[t];`
`bits(msize) data;`
`constant integer mbytes = msize DIV 8;`
`boolean first = TRUE;`
`boolean fault = FALSE;`
`boolean faulted = FALSE;`
`boolean unknown = FALSE;`

if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`

if `!AnyActiveElement(mask, esize)` then
  if `n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)` then
    `CheckSPAlignment();`
  else
    if `n == 31` then `CheckSPAlignment();`
    base = if `n == 31` then `SP[]` else `X[n];`
    offset = `Z[m];`
  for `e = 0` to `elements-1`
    if `ElemP[mask, e, esize] == '1'` then
      integer `off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);`
      bits(64) addr = base + (off << scale);
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        `data = Mem[addr, mbytes, AccType_SVE];`
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        `(data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];`
      else
        `(data, fault) = (Zeros(msize), FALSE);`
      // FFR elements set to FALSE following a suppressed access/fault
      faulted = faulted || fault;
      if faulted then
        `ElemFFR[e, esize] = '0';`
      // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
      unknown = unknown || `ElemFFR[e, esize] == '0';`
      if unknown then
        if `!fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA)` then
          `Elem[result, e, esize] = Extend(data, esize, unsigned);`
        elsif `ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO)` then
          `Elem[result, e, esize] = Zeros();`
        else // merge
          `Elem[result, e, esize] = Elem[orig, e, esize];`
        end
      else
        `Elem[result, e, esize] = Extend(data, esize, unsigned);`
      end
    end
  end
`Z[t] = result;`
LDFF1H (vector plus immediate)

Gather load first-fault unsigned halfwords to vector (immediate index)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

![32-bit element diagram]

**LDFF1H**

if !HaveSVE() then UNDEFINED;
integer t = UInit(Zt);
integer n = UInit(Zn);
integer g = UInit(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInit(imm5);

### 64-bit element

![64-bit element diagram]

**LDFF1H**

if !HaveSVE() then UNDEFINED;
integer t = UInit(Zt);
integer n = UInit(Zn);
integer g = UInit(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInit(imm5);

**Assembler Symbols**

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the base scalable vector register, encoded in the "Zn" field.
- **<imm>** Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
LDFF1SB (scalar plus scalar)

Contiguous load first-fault signed bytes to vector (scalar index)

Contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 0 1 0 | 1 1 1 | 0 |

Rm   0 1 1 | Pg | Rn | Zt |
```

dtype<3:1>dtype<0>

```
LDFF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;

32-bit element

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 0 1 0 | 1 1 0 | 1 |

Rm   0 1 1 | Pg | Rn | Zt |
```

dtype<3:1>dtype<0>

```
LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;

64-bit element

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 0 1 0 | 1 1 0 | 0 |

Rm   0 1 1 | Pg | Rn | Zt |
```

dtype<3:1>dtype<0>

```
LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
    else
        for e = 0 to elements-1
            if ElemP[mask, e, esize] == '1' then
                bits(64) addr = base + (UInt(offset) + e) * mbytes;
                if first then
                    // Mem[] will not return if a fault is detected for the first active element
                    data = Mem[addr, mbytes, AccType_SVE];
                    first = FALSE;
                else
                    // MemNF[] will return fault=TRUE if access is not performed for any reason
                    (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
                else
                    (data, fault) = (Zeros(msize), FALSE);
                    // FFR elements set to FALSE following a supressed access/fault
                    faulted = faulted || fault;
                    if faulted then
                        ElemFFR[e, esize] = '0';
                        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
                        unknown = unknown || ElemFFR[e, esize] == '0';
                    if unknown then
                        if fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                            Elem[result, e, esize] = Extend(data, esize, unsigned);
                        elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                            Elem[result, e, esize] = Zeros();
                        else
                            Elem[result, e, esize] = Elem[orig, e, esize];
                        end
                        Elem[result, e, esize] = Extend(data, esize, unsigned);
                    end
                end
            end
        end
        Z[t] = result;
```
LDFF1SB (scalar plus vector)

Gather load first-fault signed bytes to vector (vector index)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>0</td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LDFF1SB \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>0</td>
</tr>
</tbody>
</table>

LDFF1SB \{ <Zt>.S \}, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>0</td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LDFF1SB (scalar plus vector)
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>` Is the name of the offset scalable vector register, encoded in the "Zm" field.
- `<mod>` Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
    for e = 0 to elements-1
      if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
          // Mem[] will not return if a fault is detected for the first active element
          data = Mem[addr, mbytes, AccType_SVE];
          first = FALSE;
        else
          // MemNF[] will return fault=TRUE if access is not performed for any reason
          (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
          (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
        if faulted then
          ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
          if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
          elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
          else // merge
           Elem[result, e, esize] = Elem[orig, e, esize];
          else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        Z[t] = result;
LDFF1SB (vector plus immediate)

Gather load first-fault signed bytes to vector (immediate index)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   0   0   0   1   0   0   0   1   | imm5 | 1   0   1   | Pg   | Zn   | Zt   |

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   1   0   0   1   0   0   0   1   0   0   0   1   | imm5 | 1   0   1   | Pg   | Zn   | Zt   |

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
            else
                Elem[result, e, esize] = Extend(data, esize, unsigned);
        Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1SH (scalar plus scalar)

Contiguous load first-fault signed halfwords to vector (scalar index)

Contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SH { <Zt>.S }, <Pg>/Z, [ <Xn|SP>{}, <Xm>, LSL #1 ]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;

### 64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SH { <Zt>.D }, <Pg>/Z, [ <Xn|SP>{}, <Xm>, LSL #1 ]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;

Assemble Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
      end
      else
        (data, fault) = (Zeros(msize), FALSE);
      end
    end
  end
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  end
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    end
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    end
  end
  Z[t] = result;
LDFF1SH (scalar plus vector)

Gather load first-fault signed halfwords to vector (vector index)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | xs | 1 | Zm | 0 | 0 | 1 | Pg | Rn | Zt |
```

LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

32-bit unpacked scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | xs | 1 | Zm | 0 | 0 | 1 | Pg | Rn | Zt |
```

LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

32-bit unpacked unscaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | xs | 0 | Zm | 0 | 0 | 1 | Pg | Rn | Zt |
```

LDFF1SH (scalar plus vector)
LDFF1SH \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 0 |0 1| xs |0 0 1| Pg | Rn | Zt
U ff

LDFF1SH \{ <Zt>.S \}, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 0 1 0 |0 1| 1 1 | Zm | 1 0 1 | Pg | Rn | Zt
U ff

LDFF1SH \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 0 1 0 0 1 1 0 1 |0 1| Zm | 1 0 1 | Pg | Rn | Zt
msz<1>msz<0> U ff
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

| <Zt> | Is the name of the scalable vector register to be transferred, encoded in the "Zt" field. |
| <Pg> | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |
| <Xn|SP> | Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field. |
| <Zm> | Is the name of the offset scalable vector register, encoded in the "Zm" field. |
| <mod> | Is the index extend and shift specifier, encoded in "xs": |

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
  for e = 0 to elements - 1
    if ElemP[mask, e, esize] == '1' then
      integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
      bits(64) addr = base + (off << scale);
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
      else
        (data, fault) = (Zeros(msize), FALSE);
      // FFR elements set to FALSE following a supressed access/fault
      faulted = faulted || fault;
      if faulted then
        ElemFFR[e, esize] = '0';
      // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
      unknown = unknown || ElemFFR[e, esize] == '0';
      if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
          Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
          Elem[result, e, esize] = Zeros();
        else // merge
          Elem[result, e, esize] = Elem[orig, e, esize];
        else
          Elem[result, e, esize] = Extend(data, esize, unsigned);
      Z[t] = result;
LDFF1SH (vector plus immediate)

Gather load first-fault signed halfwords to vector (immediate index)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |

LDFF1SH {<Zt>.S}, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |

LDFF1SH {<Zt>.D}, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn> Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the “imm5” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
  base = Z[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    end
  else
    (data, fault) = (Zeros(msize), FALSE);
  end
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
  else
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elseif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      Elem[result, e, esize] = Elem[orig, e, esize];
    end
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  end
Z[t] = result;
```
LDFF1SW (scalar plus scalar)

Contiguous load first-fault signed words to vector (scalar index)

Contiguous load with first-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Rm</td>
</tr>
<tr>
<td>28</td>
<td>Pg</td>
</tr>
<tr>
<td>27</td>
<td>Rn</td>
</tr>
<tr>
<td>31</td>
<td>Zt</td>
</tr>
</tbody>
</table>

dtype<3:1> dtype<0>

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
      else
        (data, fault) = (Zeros(msize), FALSE);
      // FFR elements set to FALSE following a supressed access/fault
      faulted = faulted || fault;
      if faulted then
        ElemFFR[e, esize] = '0';
      // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
      unknown = unknown || ElemFFR[e, esize] == '0';
      if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNDATA) then
          Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNSZERO) then
          Elem[result, e, esize] = Zeros();
        else // merge
          Elem[result, e, esize] = Elem[orig, e, esize];
        else
          Elem[result, e, esize] = Extend(data, esize, unsigned);
      Z[t] = result;
LDFF1SW (scalar plus vector)

Gather load first-fault signed words to vector (vector index)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit unpacked scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>
```

```
LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]
```

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

32-bit unpacked unscaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>
```

```
LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>
```

```
LDFF1SW (scalar plus vector)
LDFF1SW \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

LDFF1SW \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

\[
\begin{array}{|c|c|}
\hline
xs & <mod> \\
\hline
0 & UXTW \\
1 & SXTW \\
\hline
\end{array}
\]
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      integer off = Int(Elem[offset, e, esize]<offs size-1:0>, offs unsigned);
      bits(64) addr = base + (off << scale);
      if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
      else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
      else
        (data, fault) = (Zeros(msize), FALSE);
      // FFR elements set to FALSE following a suppressed access/fault
      faulted = faulted || fault;
      if faulted then
        ElemFFR[e, esize] = '0';
      // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
      unknown = unknown || ElemFFR[e, esize] == '0';
      if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
          Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
          Elem[result, e, esize] = Zeros();
        else // merge
          Elem[result, e, esize] = Elem[orig, e, esize];
        else
          Elem[result, e, esize] = Extend(data, esize, unsigned);
      Z[t] = result;
LDFF1SW (vector plus immediate)

Gather load first-fault signed words to vector (immediate index)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits[PL] mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault & ConstrainsUnPredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrainsUnPredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else
                Elem[result, e, esize] = Elem[orig, e, esize];
            else
                Elem[result, e, esize] = Extend(data, esize, unsigned);
        Z[t] = result;
**LDFF1W (scalar plus scalar)**

Contiguous load first-fault unsigned words to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |
| Rm | 0  | 1  | 1  | Pg | Rn | Zt |

dtype<3:1>dtype<0>

**LDFF1W** `{ <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |
| Rm | 0  | 1  | 1  | Pg | Rn | Zt |

dtype<3:1>dtype<0>

**LDFF1W** `{ <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for e = 0 to elements-1
if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    if first then
        // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_SVE];
        first = FALSE;
    else
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
LDFF1W (scalar plus vector)

Gather load first-fault unsigned words to vector (vector index)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| 1 0 0 0 0 1 0 1 0 xs | 1 | Zm | 0 1 1 | Pg | Rn | Zt |

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>], <Zm>.S, <mod> #2

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| 1 1 0 0 0 1 0 1 0 xs | 1 | Zm | 0 1 1 | Pg | Rn | Zt |

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>], <Zm>.D, <mod> #2

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| 1 1 0 0 0 1 0 | 1 | 0 | xs | 0 | Zm | 0 1 1 | Pg | Rn | Zt |

LDFF1W (scalar plus vector)
LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
</tr>
<tr>
<td>U ff</td>
</tr>
</tbody>
</table>

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>U ff</td>
</tr>
</tbody>
</table>

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt; U ff</td>
</tr>
</tbody>
</table>
LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = Uint(Zt);
integer n = Uint(Rn);
integer m = Uint(Zm);
integer g = Uint(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    bits(64) addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_SVE];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
LDFF1W (vector plus immediate)

Gather load first-fault unsigned words to vector (immediate index)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. It has encodings from 2 classes: 32-bit element and 64-bit element.

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | Pg | Zn | Zt |
| msz<1> | msz<0> | U | ff |

LDFF1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | Pg | Zn | Zt |
| msz<1> | msz<0> | U | ff |

LDFF1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #imm}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn> Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem;base, e, esize), 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNF1B

Contiguous load non-fault unsigned bytes to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

dtype<3:1> dtype<0>

LDNF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

16-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

dtype<3:1> dtype<0>

LDNF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

dtype<3:1> dtype<0>

LDNF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
**64-bit element**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

```
dtype<3:1>dtype<0>
```

**LDNF1B** \{ \texttt{<Zt>.D}, \texttt{<Pg>/Z}, [\texttt{<Xn|SP>}, \\texttt{#<imm>}, \texttt{MUL VL}] \}

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

**Assembler Symbols**

- \texttt{<Zt>}: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- \texttt{<Pg>}: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \texttt{<Xn|SP>}: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- \texttt{<imm>}: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Contiguous load non-fault doublewords to vector (immediate index)

Contiguous load with non-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrunpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrunpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrunpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else  // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t] = result;
```
LDNF1H

Contiguous load non-fault unsigned halfwords to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | imm4| 1 | 0 | 1 | Pg | Rn | Zt |

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | imm4| 1 | 0 | 1 | Pg | Rn | Zt |

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

64-bit element

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | imm4| 1 | 0 | 1 | Pg | Rn | Zt |

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrinUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
            base = if n == 31 then SP[ ] else X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINTED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault & ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrinUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
LDNF1SB

Contiguous load non-fault signed bytes to vector (immediate index)

Contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  |

LDNF1SB \{ <Zt>.H }, <Pg>/Z, [<Xn|SP>{{, #<imm>, MUL VL}}]

if \(!HaveSVE()\) then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

32-bit element

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  |

LDNF1SB \{ <Zt>.S }, <Pg>/Z, [<Xn|SP>{{, #<imm>, MUL VL}}]

if \(!HaveSVE()\) then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  |

LDNF1SB \{ <Zt>.D }, <Pg>/Z, [<Xn|SP>{{, #<imm>, MUL VL}}]

if \(!HaveSVE()\) then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault & ConstrainingUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elseif ConstrainingUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
```
LDNF1SH

Contiguous load non-fault signed halfwords to vector (immediate index)

Contiguous load with non-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | imm4 | 1  | 0  | 1  | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LDNF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm4 | 1  | 0  | 1  | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LDNF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
  else
    for e = 0 to elements-1
      if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
      else
        (data, fault) = (Zeros(msize), FALSE);
      // FFR elements set to FALSE following a supressed access/fault
      faulted = faulted || fault;
      if faulted then elemFFR[e, esize] = '0';
      // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
      unknown = unknown || elemFFR[e, esize] == '0';
      if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
          Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
          Elem[result, e, esize] = Zeros();
        else // merge
          Elem[result, e, esize] = Elem[orig, e, esize];
      else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      Z[t] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNF1SW

Contiguous load non-fault signed words to vector (immediate index)

Contiguous load with non-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

### Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```
Operation

CheckSVEEnabled();
in{teger elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else  // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t] = result;
**LDNF1W**

Contiguous load non-fault unsigned words to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDNF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

### 64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDNF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
// FFR elements set to FALSE following a supressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros();
  else // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t] = result;
```
LDNT1B (scalar plus immediate)

Contiguous load non-temporal bytes to vector (immediate index)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
else
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements) + e;
            bits(64) addr = base + eoff * mbytes;
            Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
        else
            Elem[result, e, esize] = Zeros();
    Z[t] = result;
LDNT1B (scalar plus scalar)

Contiguous load non-temporal bytes to vector (scalar index)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

![ASM](image)

LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '1111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LDNT1D (scalar plus immediate)

Contiguous load non-temporal doublewords to vector (immediate index)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

\[
\text{LDNT1D} \{ <Zt>.D \}, <Pg>/\text{Z}, [<Xn|SP>\{, #<imm>, MUL VL}\}]
\]

\[
\text{if } \neg \text{HaveSVE}() \text{ then UNDEFINED;}
\]
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);

Assembler Symbols

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

\[
\text{CheckSVEEnabled}();
\]
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if \neg \text{AnyActiveElement}(mask, esize) then
\]
if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
\]
CheckSPAlignment();
\]
else
\]
if n == 31 then CheckSPAlignment();
base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
\]
if Elem[mask, e, esize] == '1' then
\]
integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
\]
else
\]
Elem[result, e, esize] = Zeros();
\]
Z[t] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNT1D (scalar plus scalar)

Contiguous load non-temporal doublewords to vector (scalar index)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector. A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 1 0 0 Rm 1 1 0 Pg Rn Zt
msz<1>msz<0>

LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
  for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
      Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
      Elem[result, e, esize] = Zeros();
  Z[t] = result;
LDNT1H (scalar plus immediate)

Contiguous load non-temporal halfwords to vector (immediate index)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

```
LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAignment();
else
  if n == 31 then CheckSPAignment();
  base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Assembler Symbols

- `<Zt>`
- `<Pg>`
- `<Xn|SP>`
- `<imm>`
LDNT1H (scalar plus scalar)

Contiguous load non-temporal halfwords to vector (scalar index)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPA(ualignment());
    else
        if n == 31 then CheckSPA(ualignment());
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1'
            then
                bits(64) addr = base + (UInt(offset) + e) * mbytes;
                Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
            else
                Elem[result, e, esize] = Zeros();
        Z[t] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNT1W (scalar plus immediate)

Contiguous load non-temporal words to vector (immediate index)

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

msz<1>msz<0>

LDNT1W \{ <Zt>.S \}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then 
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then 
    CheckSPAlignment();
  else 
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then 
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNT1W (scalar plus scalar)

Contiguous load non-temporal words to vector (scalar index)

Contiguous load non-temporal words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
        for e = 0 to elements-1
            if ElemP[mask, e, esize] == '1' then
                bits(64) addr = base + (UInt(offset) + e) * mbytes;
                Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
            else
                Elem[result, e, esize] = Zeros();
        Z[t] = result;
LDR (predicate)

Load predicate register

Load a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The load is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.

```
LDR <Pt>, [<Xn|SP>], #<imm>, MUL VL]
```

Assembler Symbols

- `<Pt>` Is the name of the destination scalable predicate register, encoded in the "Pt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

```
CheckSVEEnabled();
integer elements = PL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(Pt) result;
if n == 31 then
  CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_SVE, FALSE);
for e = 0 to elements-1
  Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned];
  offset = offset + 1;
P[t] = result;
```
LDR (vector)

Load vector register

Load a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.
The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

LDR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(VL) result;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SVE, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned];
    offset = offset + 1;
Z[t] = result;
LSL (immediate, predicated)

Logical shift left by immediate (predicated)

Shift left by immediate each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|   | L  | U  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSL(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**LSL (immediate, unpredicated)**

Logical shift left by immediate (unpredicated)

Shift left by immediate each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 0 | tzh | 1 | tszl | imm3 | 1 | 0 | 0 | 1 | 1 | 1 | Zn | Zd |

**LSL <Zd>.<T>, <Zn>.<T>, #<const>**

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tzh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tzh:tszl":

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = LSL(element1, shift);
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12, rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSL (vectors)

Logical shift left by vector (predicated)

Shift left active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 0 1 0 0 | size  0 1 0 0 1 1 0 0 | Pg  Zm  Zdn
```

\[ \text{LSL} <\text{Zdn}>, <\text{T}>; <\text{Pg}>/M, <\text{Zdn}>, <\text{T}>; <\text{Zm}>, <\text{T}> \]

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size”:
  ```plaintext
  \[
  \begin{array}{cc}
  \text{size} & \text{<T>} \\
  00 & B \\
  01 & H \\
  10 & S \\
  11 & D \\
  \end{array}
  \]
  ```
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSL(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSL (wide elements, predicated)

Logical shift left by 64-bit wide elements (predicated)

Shift left active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> <T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;

for e = 0 to elements-1
   if ElemP[mask, e, esize] == '1' then
      bits(esize) element1 = Elem[operand1, e, esize];
      bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
      integer shift = Min(UInt(element2), esize);
      Elem[result, e, esize] = LSL(element1, shift);
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];

   Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSL (wide elements, unpredicated)

Logical shift left by 64-bit wide elements (unpredicated)

Shift left all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

![Shift left matrix]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | Zn | Zm | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

LSL <Zd>.<T>, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
  integer shift = Min(UInt(element2), esize);
  Elem[result, e, esize] = LSL(element1, shift);
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSLR

Reversed logical shift left by vector (predicated)

Reversed shift left active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LSLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = LSL(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**LSR (immediate, predicated)**

Logical shift right by immediate (predicated)

Shift right by immediate, inserting zeroes, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

---

### Assembler Symbols

- **<Zdn>**
  - Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

- **<T>**
  - Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>**
  - Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- **<const>**
  - Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

---

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSR (immediate, unpredicated)

Logical shift right by immediate (unpredicated)

Shift right by immediate, inserting zeroes, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

LSR <Zd>.<T>, <Zn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = LSR(element1, shift);
Z[d] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSR (vectors)

Logical shift right by vector (predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|---------------|---------------|---------------|
| 0  0  0  0  1  0  0 | 0  1  0  0  1  1  0  0 | Pg | Zm | Zdn |
```

LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LSR (wide elements, predicated)

Logical shift right by 64-bit wide elements (predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector.

The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

### Assembler Symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSR (wide elements, unpredicated)

Logical shift right by 64-bit wide elements (unpredicated)

Shift right, inserting zeroes, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

```
0 0 0 0 0 1 0 0 | size | 1
0 1 0 0 0 0 1 | Zm | 1
0 0 0 0 0 0 0 | Zn | 0
```

LSR <Zd>.<T>, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
  integer shift = Min(UInt(element2), esize);
  Elem[result, e, esize] = LSR(element1, shift);
Z[d] = result;
```
**LSRR**

Reversed logical shift right by vector (predicated)

Reversed shift right, inserting zeroes, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  1  0  0 | size | 0  1  0 | 1  0 | 1  1  0  0 | Pg | Zm | Zdn
```

**LSRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>**

```java
if (!HaveSVE()) then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size | <T>
  - 00 | B
  - 01 | H
  - 10 | S
  - 11 | D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    Elem[result, e, esize] = LSR(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Multiply-add vectors (predicated), writing multiplicand \( Z_{dn} = Z_a + Z_{dn} \times Z_m \)

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

MAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = FALSE;

**Assembler Symbols**

- \(<Zdn>\) is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- \(<T>\) is the size specifier, encoded in "size":
  - \(<T>\) size
  - 00: B
  - 01: H
  - 10: S
  - 11: D
- \(<Pg>\) is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Zm>\) is the name of the second source scalable vector register, encoded in the "Zm" field.
- \(<Za>\) is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Multiply-add vectors (predicated), writing addend [Zda = Zda + Zn * Zm]

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer product = element1 * element2;
        if sub_op then
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Multiply-subtract vectors (predicated), writing addend [Zda = Zda - Zn * Zm]

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```mls
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
if Elem[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];
Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV

Move logical bitmask immediate to vector (unpredicated)

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This is an alias of DUPM. This means:

- The encodings in this description are named to match the encodings of DUPM.
- The description of DUPM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

MOV <Zd>.<T>, #<const>

is equivalent to

DUPM <Zd>.<T>, #<const>

and is the preferred disassembly when SVEMoveMaskPreferred(imm13).

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Operation

The description of DUPM gives the operational pseudocode for this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV

Move predicate (unpredicated)

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.

This is an alias of ORR (predicates). This means:

- The encodings in this description are named to match the encodings of ORR (predicates).
- The description of ORR (predicates) gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

MOV <Pd>.B, <Pn>.B

is equivalent to


and is the preferred disassembly when $S = '0' \land \text{Pn} = \text{Pm} \land \text{Pm} = \text{Pg}$.

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ORR (predicates) gives the operational pseudocode for this instruction.
MOV (immediate, predicated, merging)

Move signed integer immediate to vector elements (merging)

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

As an alias of CPY (immediate, merging), MOV (immediate, predicated, merging) is equivalent to

```assembly
MOV <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}
```

is equivalent to

```assembly
CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":
  ```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<imm>` Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":
  ```
<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>
  ```

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (immediate, predicated, zeroing)

Move signed integer immediate to vector elements (zeroing)

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is “#<simm8>, LSL #8”. However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as “#0, LSL #8”.

This is an alias of CPY (immediate, zeroing). This means:
- The encodings in this description are named to match the encodings of CPY (immediate, zeroing).
- The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & 1 & \text{Pg} & 0 & 0 & \text{sh} & \text{imm8} & \text{Zd} \\
\end{array}
\]

MOV <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the “Pg” field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.
MOV (immediate, unpredicated)

Move signed immediate to vector elements (unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The description of DUP (immediate) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

MOV <Zd>.<T>, #<imm>{, <shift>}

is equivalent to

DUP <Zd>.<T>, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.
MOV (predicate, predicated, merging)

Move predicates (merging)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. Does not set the condition flags.

This is an alias of SEL (predicates). This means:

- The encodings in this description are named to match the encodings of SEL (predicates).
- The description of SEL (predicates) gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

MOV <Pd>.B, <Pg>/M, <Pn>.B

is equivalent to


and is the preferred disassembly when Pd == Pm.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of SEL (predicates) gives the operational pseudocode for this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV (predicate, predicated, zeroing)

Move predicates (zeroing)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of AND (predicates). This means:

• The encodings in this description are named to match the encodings of AND (predicates).
• The description of AND (predicates) gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | Pm | 0  | 1  | Pg | 0  | Pn | 0  | Pd |

MOV <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when S == '0' && Pn == Pm.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of AND (predicates) gives the operational pseudocode for this instruction.
**MOV (scalar, predicated)**

Move general-purpose register to vector elements (predicated)

Move the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of **CPY (scalar)**. This means:

- The encodings in this description are named to match the encodings of **CPY (scalar)**.
- The description of **CPY (scalar)** gives the operational pseudocode for this instruction.

### Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<R>** Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

- **<n|SP>** Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

### Operation

The description of **CPY (scalar)** gives the operational pseudocode for this instruction.

### Operational information

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The **MOVPRFX** instruction must conform to all of the following requirements, otherwise the behavior of the **MOVPRFX** and this instruction is UNPREDICTABLE:

- The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The **MOVPRFX** instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (scalar, unpredicated)

Move general-purpose register to vector elements (unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (scalar). This means:

- The encodings in this description are named to match the encodings of DUP (scalar).
- The description of DUP (scalar) gives the operational pseudocode for this instruction.

$\text{MOV } <\text{Zd}>, <\text{T}>, <\text{R}><\text{n}\mid\text{SP}>$

is equivalent to

$\text{DUP } <\text{Zd}>, <\text{T}>, <\text{R}><\text{n}\mid\text{SP}>$

and is always the preferred disassembly.

Assembler Symbols

$<\text{Zd}>$ Is the name of the destination scalable vector register, encoded in the "Zd" field.

$<\text{T}>$ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<\text{R}>$ Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

$<\text{n}\mid\text{SP}>$ Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

The description of DUP (scalar) gives the operational pseudocode for this instruction.
MOV (SIMD&FP scalar, predicated)

Move SIMD&FP scalar register to vector elements (predicated)

Move the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (SIMD&FP scalar). This means:

- The encodings in this description are named to match the encodings of CPY (SIMD&FP scalar).
- The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{Pg} & \text{Vn} & \text{Zd} \\
\end{array}
\]

MOV <Zd>.<T>, <Pg>/M, <V><n>

is equivalent to

CPY <Zd>.<T>, <Pg>/M, <V><n>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**MOV (SIMD&FP scalar, unpredicated)**

Move indexed element or SIMD&FP scalar to vector (unpredicated)

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of **DUP (indexed)**. This means:

- The encodings in this description are named to match the encodings of **DUP (indexed)**.
- The description of **DUP (indexed)** gives the operational pseudocode for this instruction.

```
 0 0 0 0 1 0 1|imm2|1|  tsz | 0 0 1 0 0 0 |  Zn |  Zd
```

**MOV <Zd>.<T>, <Zn>.<T>[<imm>]**

is equivalent to

**DUP <Zd>.<T>, <Zn>.<T>[<imm>]**

and is the preferred disassembly when BitCount(imm2:tsz) > 1.

**MOV <Zd>.<T>, <V><n>**

is equivalent to

**DUP <Zd>.<T>, <Zn>.<T>[0]**

and is the preferred disassembly when BitCount(imm2:tsz) == 1.

**Assembler Symbols**

- **<Zd>** is the name of the destination scalable vector register, encoded in the “Zd” field.
- **<T>** is the size specifier, encoded in “tsz”:
  ```
  tsz  |  <T>  |
  00000  |  RESERVED  |
  xxxx1  |   B       |
  xxx10  |   H       |
  xx100  |   S       |
  x1000  |   D       |
  10000  |   Q       |
  ```
- **<Zn>** is the name of the source scalable vector register, encoded in the “Zn” field.
- **<imm>** is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in “imm2:tsz”.
- **<V>** is a width specifier, encoded in “tsz”:
  ```
  tsz   |  <V>  |
  00000  |  RESERVED  |
  xxxx1  |   B       |
  xxx10  |   H       |
  xx100  |   S       |
  x1000  |   D       |
  10000  |   Q       |
  ```
- **<n>** is the number [0-31] of the source SIMD&FP register, encoded in the “Zn” field.

**Operation**

The description of **DUP (indexed)** gives the operational pseudocode for this instruction.
MOV (SIMD & FP scalar, unpredicated)
MOV (vector, predicated)

Move vector elements (predicated)

Move elements from the source vector to the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of SEL (vectors). This means:

- The encodings in this description are named to match the encodings of SEL (vectors).
- The description of SEL (vectors) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 0 0 0 0 1 1 | size | Zm | 1 1 | Pg | Zn | Zd |

MOV <Zd>.<T>, <Pg>/M, <Zn>.<T>

is equivalent to

SEL <Zd>.<T>, <Pg>, <Zn>.<T>, <Zd>.<T>

and is the preferred disassembly when Zd == Zm.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

The description of SEL (vectors) gives the operational pseudocode for this instruction.
MOV (vector, unpredicated)

Move vector register (unpredicated)

Move vector register. This instruction is unpredicated.

This is an alias of ORR (vectors, unpredicated). This means:

- The encodings in this description are named to match the encodings of ORR (vectors, unpredicated).
- The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

\[
\begin{array}{ccccccccccccc}
\end{array}
\]

\begin{array}{ccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & \text{Zm} & 0 & 0 & 1 & 1 & 0 & 0 & \text{Zn} & & & & & & & & & & & & & & \text{Zd}
\end{array}

MOV \langle \text{Zd} \rangle .D, \langle \text{Zn} \rangle .D

is equivalent to

ORR \langle \text{Zd} \rangle .D, \langle \text{Zn} \rangle .D, \langle \text{Zn} \rangle .D

and is the preferred disassembly when Zn == Zm.

Assembler Symbols

\langle \text{Zd} \rangle \quad \text{Is the name of the destination scalable vector register, encoded in the "Zd" field.}
\langle \text{Zn} \rangle \quad \text{Is the name of the first source scalable vector register, encoded in the "Zn" field.}

Operation

The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.
**MOVPRFX (predicated)**

Move prefix (predicated)

The predicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit "wide vector" operand), and the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | M  | 0  | 0  | 1  | Pg | Zn | Zd |

**MOVPRFX** `<Zd>`, `<Pg>/<ZM>`, `<Zn>`<T>

if `!HaveSVE()` then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean merging = (M == '1');

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in "M":

<table>
<thead>
<tr>
<th>M</th>
<th>&lt;ZM&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

**CheckSVEEnabled();**
integer elements = \( \text{VL} \) DIV esize;
bits(\( \text{PL} \)) mask = \( \text{P} \)[g];
bits(\( \text{VL} \)) operand1 = if AnyActiveElement(mask, esize) then \( \text{Z} \)[n] else Zeros();
bits(\( \text{VL} \)) dest = \( \text{Z} \)[d];
bits(\( \text{VL} \)) result;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand1, e, esize];
    Elem[result, e, esize] = element;
  elsif merging then
    Elem[result, e, esize] = Elem[dest, e, esize];
  else
    Elem[result, e, esize] = Zeros();
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
The unpredicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Although the operation of the instruction is defined as a simple unpredicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

```
MOVPRFX <Zd>, <Zn>
```

```
if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler Symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```
CheckSVEEnabled();
bits(VL) result = Z[n];
Z[d] = result;
```

Internal version only: isa v33.16decorel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOVS (predicated)

Move predicates (zeroing), setting the condition flags

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of ANDS. This means:

- The encodings in this description are named to match the encodings of ANDS.
- The description of ANDS gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pm</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>0</td>
<td>Pn</td>
<td>0</td>
<td>Pd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S

MOVS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when S == '1' \&\& Pn == Pm.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ANDS gives the operational pseudocode for this instruction.
**MOVS (unpredicated)**

Move predicate (unpredicated), setting the condition flags

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the `FIRST (N)` , `NONE (Z)` , `!LAST (C)` condition flags based on the predicate result, and the `V` flag to zero.

This is an alias of **ORRS**. This means:

- The encodings in this description are named to match the encodings of **ORRS**.
- The description of **ORRS** gives the operational pseudocode for this instruction.

```
0 0 1 0 1 0 1 1 1 0 0 | Pm | 0 1 | Pg | 0 | Pn | 0 | Pd
S
```

MOVS `<Pd>.B, <Pn>.B`

is equivalent to


and is the preferred disassembly when `S == '1' && Pn == Pm && Pm == Pg`.

**Assembler Symbols**

- `<Pd>`: Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pn>`: Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **ORRS** gives the operational pseudocode for this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Multiply-subtract vectors (predicated), writing multiplicand \([Z_{dn} = Z_a - Z_{dn} \times Z_m]\)

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = TRUE;
```

**Assembler Symbols**

\(<Z_{dn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Z_{m}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

\(<Z_{a}>\) Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

```
CheckSVEEnabled();
integer elements = _VL_ DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MUL (immediate)

Multiply by immediate (unpredicated)

Multiply by an immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
0 0 1 0 0 1 0 1 | size | 1 1 0 | 0 0 | 0 1 1 0 | imm8 | Zdn
```

MUL <Zdn>, <T>, <Zdn>.<T>, #<imm>

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = SInt(imm8);
```

Assembler Symbols

- `<Zdn>`: Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>`: Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (element1 * imm)<esize-1:0>;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MUL (vectors)

Multiply vectors (predicated)

Multiply active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

### Assembler Symbols

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

if !HaveSVE() then UNDEFINED;
integer elements = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

Zdn: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

T: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th></th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Pg: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Zm: Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer elements = 8 << UInt(size);
integer g = UInt(Pg);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

if !HaveSVE() then UNDEFINED;
integer elements = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer product = element1 * element2;
    Elem[result, e, esize] = product<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
NAND

Bitwise NAND predicates

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & P_m & 0 & 1 & P_g & 1 & P_n & 1 & P_d
\end{array}
\]

NAND \langle P_d \rangle .B, \langle P_g \rangle /Z, \langle P_n \rangle .B, \langle P_m \rangle .B

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8;
integer g = \text{UInt}(P_g);
integer n = \text{UInt}(P_n);
integer m = \text{UInt}(P_m);
integer d = \text{UInt}(P_d);
boolean setflags = FALSE;

Assembler Symbols

\langle P_d \rangle Is the name of the destination scalable predicate register, encoded in the "Pd" field.
\langle P_g \rangle Is the name of the governing scalable predicate register, encoded in the "Pg" field.
\langle P_n \rangle Is the name of the first source scalable predicate register, encoded in the "Pn" field.
\langle P_m \rangle Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

\text{CheckSVEEnabled}();
integer elements = \text{VL} \div \text{esize};
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = \text{ElemP}[\text{operand1}, e, \text{esize}];
    bit element2 = \text{ElemP}[\text{operand2}, e, \text{esize}];
    if \text{ElemP}[mask, e, \text{esize}] == '1'
        then
            \text{ElemP}[result, e, \text{esize}] = \text{NOT}(element1 \text{ AND } element2);
        else
            \text{ElemP}[result, e, \text{esize}] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = \text{PredTest}(\text{mask, result, \text{esize}});
    P[d] = result;
NANDS

Bitwise NAND predicates, setting the condition flags

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
Negate (predicated)

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector remain unmodified.

```
NEG <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

If `!HaveSVE()` then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand, e, esize]);
    element = -element;
    Elem[result, e, esize] = element<esize-1:0>;
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
NOR

Bitwise NOR predicates

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

```
0 0 1 0 0 1 1 0 0 | Pm | 0 1 | Pg | 1 | Pn | 0 | Pd
```


```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```
NORS

Bitwise NOR predicates, setting the condition flags

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 0 1 0 0 1 0 1 1 0 0 | Pm | 0 1 | Pg | 1 | Pn | 0 | Pd |


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
NOT (predicate)

Bitwise invert predicate

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of EOR (predicates). This means:

- The encodings in this description are named to match the encodings of EOR (predicates).
- The description of EOR (predicates) gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | Pm  | 0   | 1   | Pg  | 1   | Pn  | 0   | Pd  |
```

NOT <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when Pm == Pg.

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of EOR (predicates) gives the operational pseudocode for this instruction.
NOT (vector)

Bitwise invert vector (predicated)

Bitwise invert each active element of the source vector, and place the results in the corresponding elements of the
destination vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & \text{Pg} & \text{Zn} & \text{Zd}
\end{array}
\]

\text{NOT } <\text{Zd}>.<<T>>, <\text{Pg}>/M, <\text{Zn}>.<<T>>

if \text{!HaveSVE()} then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer n = \text{UInt}(\text{Zn});
integer d = \text{UInt}(\text{Zd});

\textbf{Assembler Symbols}

\begin{itemize}
  \item \textbf{<Zd>} Is the name of the destination scalable vector register, encoded in the “Zd” field.
  \item \textbf{<T>} Is the size specifier, encoded in “size”:
    \begin{tabular}{c|c}
      size & <T> \\
      \hline
      00 & B \\
      01 & H \\
      10 & S \\
      11 & D
    \end{tabular}
  \item \textbf{<Pg>} Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
  \item \textbf{<Zn>} Is the name of the source scalable vector register, encoded in the “Zn” field.
\end{itemize}

\textbf{Operation}

\textbf{CheckSVEEnabled()};
integer elements = \text{VL} \text{DIV} esize;
bits(PL) mask = P[g];
bits(VL) operand = if \text{AnyActiveElement}(mask, esize) then Z[n] else \text{Zeros}();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if \text{ElemP}[mask, e, esize] == '1' then
    bits(esize) element = \text{Elem}[operand, e, esize];
    \text{Elem}[result, e, esize] = \text{NOT} element;
\text{Z}[d] = result;

\textbf{Operational information}

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
\begin{itemize}
  \item The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
       and source element size as this instruction.
  \item The MOVPRFX instruction must specify the same destination register as this instruction.
  \item The destination register must not refer to architectural register state referenced by any other source operand
       register of this instruction.
\end{itemize}
NOTS

Bitwise invert predicate, setting the condition flags

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of EORS. This means:

• The encodings in this description are named to match the encodings of EORS.
• The description of EORS gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | Pm | 0  | 1  | Pg | 1  | Pn | 0  | Pd |

NOTS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when Pm == Pg.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of EORS gives the operational pseudocode for this instruction.
ORN (immediate)

Bitwise inclusive OR with inverted immediate (unpredicated)

Bitwise inclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

```
ORN <Zdn>.<T>, <Zdn>.<T>, #<const>
```

is equivalent to

```
ORR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)
```

Assembler Symbols

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<const>` is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ORN (predicates)

Bitwise inclusive OR inverted predicate

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 1 1 0 0 0 1 1 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1

ORN \(<Pd>\.B, <Pg>/Z, <Pn>\.B, <Pm>\.B\)

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler Symbols

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.
\(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.
\(<Pm>\) Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

\(\text{CheckSVEEnabled}()\);
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 OR (NOT element2);
  else
    ElemP[result, e, esize] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
Bitwise inclusive OR inverted predicate, setting the condition flags

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Assembler Symbols

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` is the name of the first source scalable predicate register, encoded in the "Pn" field.
- `<Pm>` is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```c
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR (NOT element2);
    else
        ElemP[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```
ORR (immediate)

Bitwise inclusive OR with immediate (unpredicated)

Bitwise inclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction ORN (immediate).

```
0 0 0 0 0 1 0 1 | 0 0 0 0 0 0 | imm13        | Zdn
```

**Operation**

```
if !HaveSVE() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

integer elements = VL DIV 64;
bits(VL) operand = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 OR imm;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

---

**Assembler Symbols**

- `<Zdn>`  
  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

- `<T>`  
  Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<const>`  
  Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.
ORR (predicates)

Bitwise inclusive OR predicates

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is used by the alias MOV.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>S == '0' &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1 OR element2;
  else
    ElemP[result, e, esize] = '0';
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ORR (vectors, predicated)

Bitwise inclusive OR vectors (predicated)

Bitwise inclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 0 | size | 0 1 1 | 0 0 0 0 0 0 0 | Pg | Zm | Zdn |

ORR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size  <T>
    - 00  B
    - 01  H
    - 10  S
    - 11  D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element1 OR element2;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**ORR (vectors, unpredicated)**

Bitwise inclusive OR vectors (unpredicated)

Bitwise inclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the first in the corresponding elements of the destination vector. This instruction is unpredicated. This instruction is used by the alias **MOV (vector, unpredicated)**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | Zm | 0  | 0  | 1  | 1  | 0  | 0  | Zn | Zd |

**ORR <Zd>.D, <Zn>.D, <Zm>.D**

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, unpredicated)</td>
<td>Zn == Zm</td>
</tr>
</tbody>
</table>

**Operation**

`CheckSVEEnabled();`

`bits(VL) operand1 = Z[n];`

`bits(VL) operand2 = Z[m];`

`Z[d] = operand1 OR operand2;`

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ORRS

Bitwise inclusive OR predicates, setting the condition flags

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the alias MOVS (unpredicated).

![Predicate Register](image)

**ORRS \(<Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B\)**

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

**Assembler Symbols**

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

\(<Pm>\) Is the name of the second source scalable predicate register, encoded in the "Pm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVS (unpredicated)</td>
<td>(S == '1' &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg)</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
   bit element1 = ElemP[operand1, e, esize];
   bit element2 = ElemP[operand2, e, esize];
   if ElemP[mask, e, esize] == '1' then
      ElemP[result, e, esize] = element1 OR element2;
   else
      ElemP[result, e, esize] = '0';
if setflags then
   PSTATE.\<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
Bitwise inclusive OR reduction to scalar

Bitwise inclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

|    31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| size 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 | Pg 0 0 0 0 0 1 Zn | Vd 0 0 0 0 0 1 |

ORV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

| <V> | Is a width specifier, encoded in “size”:
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

| <d> | Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field. |
|-----|
| <Pg> | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |
|-----|
| <Zn> | Is the name of the source scalable vector register, encoded in the "Zn" field. |
|-----|
| <T> | Is the size specifier, encoded in “size”:
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(esize) result = Zeros(esize);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        result = result OR Elem[operand, e, esize];
V[d] = result;
**PFALSE**

Set all predicate elements to false

Set all elements in the destination predicate to false.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

**PFALSE <Pd>.B**

if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);

**Assembler Symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

**Operation**

```cpp
CheckSVEEnabled();
P[d] = Zeros(PL);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Set the first active predicate element to true

Sets the first active element in the destination predicate to true, otherwise elements from the source predicate are passed through unchanged. Sets the \textsc{first} (N), \textsc{none} (Z), \textsc{!last} (C) condition flags based on the predicate result, and the V flag to zero.

\begin{verbatim}
0 0 1 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0
 Pg 0      Pdn
\end{verbatim}

PFIRST \texttt{<Pdn>.B}, \texttt{<Pg>}, \texttt{<Pdn>.B}

if \texttt{!HaveSVE()} then UNDEFINED;
integer esize = 8;
integer g = \texttt{UInt}(Pg);
integer dn = \texttt{UInt}(Pdn);

Assembler Symbols

\texttt{<Pdn>}: Is the name of the source and destination scalable predicate register, encoded in the "Pdn" field.
\texttt{<Pg>}: Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

\texttt{CheckSVEEnabled()};
integer elements = \texttt{VL} \div esize;
bits(PL) mask = \texttt{P}[g];
bits(PL) result = \texttt{P}[dn];
integer first = -1;

for e = 0 to elements-1
  if \texttt{ElemP[mask, e, esize]} == '1' && first == -1 then
    first = e;
if first >= 0 then
  \texttt{ElemP[result, first, esize]} = '1';
PSTATE.<N,Z,C,V> = \texttt{PredTest}(mask, result, esize);
P\texttt{[dn]} = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PNEXT

Find next active predicate

An instruction used to construct a loop which iterates over all active elements in a predicate. If all source predicate elements are false it sets the first active predicate element in the destination predicate to true. Otherwise it determines the next active predicate element following the last true source predicate element, and if one is found sets the corresponding destination predicate element to true. All other destination predicate elements are set to false. Sets the FIRST(N), NONE(Z), !LAST(C) condition flags based on the predicate result, and the V flag to zero.

$$\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & P & g & 0 & P & d & n
\end{array}$$

PNEXT $<$Pdn$>$..<T$>$, $<$Pg$>$, $<$Pdn$>$..<T$>$

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Pdn);

Assembler Symbols

$<$Pdn$>$ Is the name of the source and destination scalable predicate register, encoded in the "Pdn" field.

$<$T$>$ Is the size specifier, encoded in “size”:

$$\begin{array}{c|c}
\text{size} & \text{<T>} \\
00 & B \\
01 & H \\
10 & S \\
11 & D
\end{array}$$

$<$Pg$>$ Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[dn];
bits(PL) result;

integer next = LastActiveElement(operand, esize) + 1;
while next < elements && (ElemP[mask, next, esize] == '0') do
    next = next + 1;
result = Zeros();
if next < elements then
    ElemP[result, next, esize] = '1';
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn] = result;
PRFB (scalar plus immediate)

Contiguous prefetch bytes (immediate index)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

```
PRFB <prfop>, <Pg>, [<Xn|SP'>{, #<imm>, MUL VL}]
```

```assembly
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = SInt(imm6);
```

Assembler Symbols

- `<prfop>` Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<imm>` Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements - 1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFB (scalar plus scalar)

Contiguous prefetch bytes (scalar index)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
  1 0 0 0 0 1 0 | 0 | 0 | 0 | Rm | 1 1 0 | Pg | Rn | 0 | prfop
               msz<1>msz<0>

PRFB <prfop>, <Pg>, [<Xn|SP>, <Xm>]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```
CheckSVEEnabled();
integer elements = vl DIV esize;
bits(pl) mask = P[g];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then sp[] else x[n];
    offset = x[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```
PRFB (scalar plus vector)

Gather prefetch bytes (scalar plus vector)

Gather prefetch of bytes from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

32-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0 xs 1 Zm 0 0 0 0 Pg Rn 0 prfop</td>
</tr>
</tbody>
</table>

PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;

32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 xs 1 Zm 0 0 0 0 Pg Rn 0 prfop</td>
</tr>
</tbody>
</table>

PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 1 1 Zm 1 0 0 0 Pg Rn 0 prfop</td>
</tr>
</tbody>
</table>

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”: 

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFB (vector plus immediate)

Gather prefetch bytes (vector plus immediate)

Gather prefetch of bytes from the active memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

```
| 31 30 29 28 27 26 25 24 |  23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------------|-------------------------|-------------------------|
| 1 0 0 0 0 1 0           | 0 0 0 0 imm5            |
|                         | 1 1 1 Pg Zn             |
| prfop                   |                         |
```

PRFB <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```java
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
```

64-bit element

```
| 31 30 29 28 27 26 25 24 |  23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------------|-------------------------|-------------------------|
| 1 1 0 0 0 1 0           | 0 0 0 0 imm5            |
|                         | 1 1 1 Pg Zn             |
| prfop                   |                         |
```

PRFB <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```java
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>0110</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the “Zn” field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem;base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFD (scalar plus immediate)

Contiguous prefetch doublewords (immediate index)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

```
1 0 0 0 1 0 1 1 1 | imm6  0 1 1 Pg  Rn  0 | prfop
```

msz<1>msz<0>

```
PRFD <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]}
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

```
<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(LED) mask = P[g];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFD (scalar plus scalar)

Contiguous prefetch doublewords (scalar index)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

Block diagram:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>

PRFD <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 64;
integer g = Uint(Pg);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer level = Uint(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
  base = if n == 31 then SP[] else X[n];
  offset = X[m];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = UInt(offset) + e;
    bits(64) addr = base + (eoff << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
**PRFD (scalar plus vector)**

Gather prefetch doublewords (scalar plus vector)

Gather prefetch of doublewords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 8. Inactive addresses are not prefetched from memory.

The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: **32-bit scaled offset**, **32-bit unpacked scaled offset** and **64-bit scaled offset**

### 32-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | xs|1|Zm|0|1|1|Pg|Rn|0|prfop|   |

```
PRFD <prfop>, <Pg>, [Xn|SP>, <Zm>.S, <mod> #3]
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_HINT = if prfop<3> == '0' then prefetch_READ else prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | xs|1|Zm|0|1|1|Pg|Rn|0|prfop|   |

```
PRFD <prfop>, <Pg>, [Xn|SP>, <Zm>.D, <mod> #3]
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_HINT = if prfop<3> == '0' then prefetch_READ else prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1 | Zm|1|1|1|Pg|Rn|0|prfop|   |

msz<1>msz<0>

---

This document page covers the PRFD (scalar plus vector) instruction, detailing its operation and encoding details for different types of offset, including 32-bit scaled, 32-bit unpacked scaled, and 64-bit scaled offsets. It also includes code snippets demonstrating the calculation of various parameters based on the `prfop` symbol.
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Fetch_READ else Fetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
**PRFD (vector plus immediate)**

Gather prefetch doublewords (vector plus immediate)

Gather prefetch of doublewords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive addresses are not prefetched from memory.

The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 0 0 0 0 1 0 1 1 0 0 imm5 1 1 1 Pg Zn 0 prfop |
```

**PRFD** `<prfop>`, `<Pg>`, `[<Zn>.S{, #<imm>}]`

```java
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

### 64-bit element

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 0 0 0 1 0 1 1 0 0 imm5 1 1 1 Pg Zn 0 prfop |
```

**PRFD** `<prfop>`, `<Pg>`, `[<Zn>.D{, #<imm>}]`

```java
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

**Assembler Symbols**

`<prfop>` Is the prefetch operation specifier, encoded in “prfop”: 
<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
if AnyActiveElement(mask, esize) then
    base = Z[n];

for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
## PRFH (scalar plus immediate)

Contiguous prefetch halfwords (immediate index)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

### Assembler Symbols

- `<prfop>` Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<imm>` Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
**Operation**

```c
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFH (scalar plus scalar)

Contiguous prefetch halfwords (scalar index)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | Rm | 1  | 1  | 0  | Pg | Rn | 0  | prfop |

msz<1>msz<0>

PRFH <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;

integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```assembly
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFH (scalar plus vector)

Gather prefetch halfwords (scalar plus vector)

Gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

32-bit scaled offset

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
inherent offs_size = 32;
boolean offs_unsigned = (xs == '0');
inherent scale = 1;

32-bit unpacked scaled offset

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
inherent offs_size = 32;
boolean offs_unsigned = (xs == '0');
inherent scale = 1;

64-bit scaled offset

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
inherent offs_size = 32;
boolean offs_unsigned = (xs == '0');
inherent scale = 1;
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offset_size = 64;
boolean offset_unsigned = TRUE;
integer scale = 1;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFH (vector plus immediate)

Gather prefetch halfwords (vector plus immediate)

Gather prefetch of halfwords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 0 0 1 0 0 1 0 1 0 0 0 | imm5 1 1 1 | Pg | Zn | 0 | prfop |

PRFH <prfop>, <Pg>, [Zn].S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

64-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 0 0 1 0 0 1 0 0 0 | imm5 1 1 1 | Pg | Zn | 0 | prfop |

PRFH <prfop>, <Pg>, [Zn].D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:
### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**PRFW (scalar plus immediate)**

Contiguous prefetch words (immediate index)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

```
prfop, Pg, [Xn|SP]{, #imm, MUL VL}
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = SInt(imm6);

**Assembler Symbols**

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

\textbf{CheckSVEEnabled}();
integer elements = \texttt{VL} \ DIV \ esize;
b\texttt{its}（\texttt{PL}） mask = \texttt{P}[g];
b\texttt{its}（64） base;

\textbf{if} \ \textbf{AnyActiveElement} (mask, esize) \ \textbf{then}
   \textbf{base} = \textbf{if} \ n == 31 \ \textbf{then} \ \texttt{SP}[\textit{l}] \ \textbf{else} \ \texttt{X}[\textit{n}];
\textbf{for} \ e = 0 \ \textbf{to} \ \textit{elements}-1
   \textbf{if} \ \textbf{ElemP}[mask, e, esize] == '1' \ \textbf{then}
      \textbf{integer} eoff = (offset * elements) + e;
      \textbf{bits}（64） addr = base + (eoff \ \texttt{<<} \ \texttt{scale});
      \textbf{Hint_Prefetch}(addr, pref\_hint, level, stream);
PRFW (scalar plus scalar)

Contiguous prefetch words (scalar index)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 0 0 0 0 1 | 0 | 1 | 0 | 0 | Rm | 1 | 1 | 0 | Pg | Rn | 0 | prfop |

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>111x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`

`integer elements = VL DIV esize;`

`bits(PL) mask = P[g];`

`bits(64) base;`

`bits(64) offset;`

```
if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
```

```
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        `Hint_Prefetch(addr, pref_hint, level, stream);`
```
PRFW (scalar plus vector)

Gather prefetch words (scalar plus vector)

Gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

### 32-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | xs | 1  | Zm | 0  | 1  | 0  | Pg | Rn | 0  | prfop |

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop[2:1]);
boolean stream = (prfop[0] == '1');
pref_hint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | xs | 1  | Zm | 0  | 1  | 0  | Pg | Rn | 0  | prfop |

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop[2:1]);
boolean stream = (prfop[0] == '1');
pref_hint = if prfop[3] == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | xs | 1  | Zm | 1  | 1  | 0  | Pg | Rn | 0  | prfop |

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PDL3STRM</td>
</tr>
<tr>
<td>x1x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(VL) offset;
if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFW (vector plus immediate)

Gather prefetch words (vector plus immediate)

Gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory.
The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
It has encodings from 2 classes: **32-bit element** and **64-bit element**

**32-bit element**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | imm5 | 1  | 1  | 1  | Pg  | Zn  | 0  | prfop |

**PRFW** `<prfop>`, `<Pg>`, `[<Zn>.S{, #<imm>}]`

if `!HaveSVE()` then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);

**64-bit element**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | imm5 | 1  | 1  | 1  | Pg  | Zn  | 0  | prfop |

**PRFW** `<prfop>`, `<Pg>`, `[<Zn>.D{, #<imm>}]`

if `!HaveSVE()` then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);

**Assembler Symbols**

`<prfop>` is the prefetch operation specifier, encoded in "prfop":

---

PRFW (vector plus immediate)
### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
if AnyActiveElement(mask, esize) then
    base = Z[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

---

<Prfop> | <prfop>
--- | ---
0000 | PDDL1KEEP
0001 | PDDL1STRM
0010 | PDDL2KEEP
0011 | PDDL2STRM
0100 | PDDL3KEEP
0101 | PDDL3STRM
1000 | PSTL1KEEP
1001 | PSTL1STRM
1010 | PSTL2KEEP
1011 | PSTL2STRM
1100 | PSTL3KEEP
1101 | PSTL3STRM

<Prfop> is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> is the name of the base scalable vector register, encoded in the “Zn” field.

<imm> is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the “imm5” field.
PTEST

Set condition flags for predicate

Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate source register, and the V flag to zero.

```
0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 1
```

PTEST <Pg>, <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
```

Assembler Symbols

- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) result = P[n];
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Initialise predicate from named constraint

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Does not set the condition flags.

### Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

```c
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = FALSE;
bits(5) pat = pattern;
```
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;
for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Initialise predicate from named constraint and set the condition flags

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
PTRUES <Pd>,<T>{, <pattern>}
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;
```

**Assembler Symbols**

* <Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

* <T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

* <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;
for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PUNPKHI, PUNPKLO

Unpack and widen half of predicate

Unpack elements from the lowest or highest half of the source predicate and place in elements of twice their size within the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

High half

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
PUNPKHI <Pd>.H, <Pn>.B
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = TRUE;
```

Low half

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
```

```
PUNPKLO <Pd>.H, <Pn>.B
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = FALSE;
```

Assembler Symbols

```
<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.
```

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) operand = P[n];
bits(PL) result;
for e = 0 to elements-1
    ElemP[result, e, esize] = ElemP[operand, if hi then e + elements else e, esize DIV 2];
P[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RBIT

Reverse bits (predicated)

Reverse bits in each active element of the source vector, and place the results in the corresponding elements of the
destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>Pg</td>
</tr>
<tr>
<td>Zn</td>
</tr>
<tr>
<td>Zd</td>
</tr>
</tbody>
</table>

RBIT <Zd>, <T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = BitReverse(element);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RDFFR (predicated)

Return predicate of successfully loaded elements

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  |

RDFFR <Pd>.B, <Pg>/Z

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) ffr = FFR[];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d] = result;
RDFFR (unpredicated)

Read the first-fault register

Read the first-fault register (FFR) and place in the destination predicate without predication.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Pd
\end{array}
\]

RDFFR \(<Pd>\).B

if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);

Assembler Symbols

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

CheckSVEEnabled();
bits(PL) ffr = FFR[];
P[d] = ffr;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RDFFRS

Return predicate of succesfully loaded elements, setting the condition flags

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| P                                     | Pg              | Pd              |

RDFFRS  <Pd>.B,  <Pg>/Z

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) ffr = FFR[];
bits(PL) result = ffr AND mask;
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
  P[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RDVL

Read multiple of vector register size to scalar register

Multiply the current vector register size in bytes by an immediate in the range -32 to 31 and place the result in the
64-bit destination general-purpose register.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 0 1 1 1 1 1 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

RDVL $<Xd>$, #<imm>

if !HaveSVE() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler Symbols

$<Xd>$ Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
$<imm>$ Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
integer len = imm * ($VL$ DIV 8);
$X[d]$ = len<63:0>;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
REV (predicate)

Reverse all elements in a predicate

Reverse the order of all elements in the source predicate and place in the destination predicate. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 0 0 0 0 1 0 1 | size 1 1 0 1 0 0 1 0 0 0 0 0 | Pn 0 | Pd |

REV <Pd>.<T>, <Pn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer d = UInt(Pd);

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
bits(PL) operand = P[n];
bits(PL) result = Reverse(operand, esize DIV 8);
P[d] = result;
REV (vector)

Reverse all elements in a vector (unpredicated)

Reverse the order of all elements in the source vector and place in the destination vector. This instruction is
unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| size                  | Zn    | Zd    |

REV <Zd>.<T>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
bits(VL) operand = Z[n];
bits(VL) result = Reverse(operand, esize);
Z[d] = result;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**REVB, REVH, REVW**

Reverse bytes / halfwords / words within elements (predicated)

Reverse the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: Byte, Halfword and Word

**Byte**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 0 0 0 0 1 0 1 | size 1 0 0 1 0 1 0 0 | Pg Zn Zd |
```

```
REVB <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 8;
```

**Halfword**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 0 0 0 0 1 0 1 | size 1 0 0 1 0 1 1 0 0 | Pg Zn Zd |
```

```
REVH <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

```java
if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 16;
```

**Word**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 0 0 0 0 1 0 1 | size 1 0 0 1 1 0 1 0 0 | Pg Zn Zd |
```

```
REVW <Zd>.D, <Pg>/M, <Zn>.D
```

```java
if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 32;
```

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` For the byte variant: is the size specifier, encoded in "size":
  ```
For the halfword variant: is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SABD

Signed absolute difference (predicated)

Compute the absolute difference between signed integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 0 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SADDV

Signed add reduction to scalar

Signed add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first sign-extended to 64 bits. Inactive elements in the source vector are treated as zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0 | | 0  | 0  | 0  | O  | 0  | 0  | 1  | | Pg | | Zn | | Vd |

SADDV <Dd>, <Pg>, <Zn>,<T>

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Vd);

Assembler Symbols

- <Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer sum = 0;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        sum = sum + element;

V[d] = sum<63:0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF

Signed integer convert to floating-point (predicated)

Convert to floating-point from the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: 16-bit to half-precision, 32-bit to half-precision, 32-bit to single-precision, 32-bit to double-precision, 64-bit to half-precision, 64-bit to single-precision and 64-bit to double-precision

16-bit to half-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

int_U

SCVTF <Zd>.H, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to half-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

int_U

SCVTF <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Zd);
integer s_esize = 32;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to single-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

int_U
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|---|
| 0 1 1 0 0 1 0 1 1 0 1 0 0 0 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 | Pg | Zn | Zd |

SCVTF <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to half-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|---|
| 0 1 1 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 | Pg | Zn | Zd |

SCVTF <Zd>.H, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to single-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|---|
| 0 1 1 0 0 1 0 1 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 | Pg | Zn | Zd |
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

64-bit to double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |    |

SCVTF <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Zd>                  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>                  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>                  Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
    Elem[result, e, esize] = ZeroExtend(fpval);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SDIV**

Signed divide (predicated)

Signed divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

**SDIV** <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size<0>“:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer quotient;
    if element2 == 0 then
      quotient = 0;
    else
      quotient = RoundTowardsZero(Real(element1) / Real(element2));
    Elem[result, e, esize] = quotient<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDIVR

Signed reversed divide (predicated)

Signed reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

SDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if ![HaveSVE] then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 <<(UInt)(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElementP[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element1 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element2) / Real(element1));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDOT (indexed)

Signed integer indexed dot product

The signed integer indexed dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | i2 | Zm | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zn | Zda |

size<1>size<0> U


if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | i1 | Zm | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zn | Zda |

size<1>size<0> U


if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 32-bit variant: is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.
For the 64-bit variant: is the immediate index of a quadtuplet of four 16-bit elements within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SDOT (vectors)

Signed integer dot product

The signed integer dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.

```
SDOT <Zda>, <T>, <Zn>, <Tb>, <Zm>.<Tb>
```

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

**<Zda>** Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

**<T>** Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

**<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.

**<Tb>** Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
</tbody>
</table>

**<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SEL (predicates)

Conditionally select elements from two predicates

Read active elements from the first source predicate and inactive elements from the second source predicate and place in the corresponding elements of the destination predicate. Does not set the condition flags.

This instruction is used by the alias MOV (predicate, predicated, merging).

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicate, predicated, merging)</td>
<td>Pd == Pm</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1;
    else
        ElemP[result, e, esize] = element2;
P[d] = result;
**SEL (vectors)**

Conditionally select elements from two vectors.

Read active elements from the first source vector and inactive elements from the second source vector and place in the corresponding elements of the destination vector.

This instruction is used by the alias `MOV (vector, predicated)`.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 1 | 1 | Zm | 1 1 | Pg | Zn | Zd |

**SEL <Zd>.<T>, <Zi>.<Zj>, <Zk>.<Zl>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, predicated)</td>
<td>Zd == Zm</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) operand2 = if AnyActiveElement(NOT(mask), esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SETFFR

Initialise the first-fault register to all true

Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

SETFFR

if !HaveSVE() then UNDEFINED;

Operation

CheckSVEEnabled();

\( FFR[] = \text{Ones}(PL); \)
SMAX (Immediate)

Signed maximum with immediate (unpredicated)

Determine the signed maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

### Assembler Symbols

- `<Zdn>`: Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>`: Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;
Z[dn] = result;
```

### Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

---

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMAX (vectors)

Signed maximum vectors (predicated)

Determine the signed maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

\[ \text{SMAX} \left< \text{Zdn} \right>, \left< \text{T} \right>, \left< \text{Pg} \right>/M, \left< \text{Zm} \right>, \left< \text{T} \right> \]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\text{CheckSVEEnabled}();
integer elements = VL \text{DIV} esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if \text{AnyActiveElement}(mask, esize) then Z[m] else \text{Zeros}();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a \text{MOVPRFX} instruction. The \text{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \text{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \text{MOVPRFX} instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SMAXV**

Signed maximum reduction to scalar

Signed maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 0 | size | 0 0 1 0 | 0 | 0 0 1 | Pg | Zn | Vd |

**SMAXV <V><d>, <Pg>, <Zn> <T>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;

**Assembler Symbols**

- `<V>` - Is a width specifier, encoded in "size":
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` - Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
- `<Pg>` - Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` - Is the name of the source scalable vector register, encoded in the "Zn" field.

- `<T>` - Is the size specifier, encoded in "size":
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = Int(Elem[operand, e, esize], unsigned);
    maximum = Max(maximum, element);
  
V[d] = maximum<<esize-1:0>;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SMIN (immediate)**

Signed minimum with immediate (unpredicated)

Determine the signed minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 1 0 0 1 0 1 | size | 1 0 1 0 1 0 1 | imm8 | Zdn | U |
```

**SMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
    Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMIN (vectors)

Signed minimum vectors (predicated)

Determine the signed minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|----------------|----------------|
| 0 0 0 0 1 0 0 0   | 0 0 0 1 0 1 0 0 | Pg Zm Zdn     |
```

SMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMINV

Signed minimum reduction to scalar

Signed minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.

```
0 0 0 0 0 1 0 0 | size | 0 0 1 0 1 0 0 1 | Pg | Zn | Vd
```

**SMINV** `<V><d>`, `<Pg>`, `<Zn>`.<`<T>`

```cpp
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

**Assembler Symbols**

- `<V>`: Is a width specifier, encoded in "size":
  ```
  size  <V>
  00  B
  01  H
  10  S
  11  D
  ```

- `<d>`: Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

- `<T>`: Is the size specifier, encoded in "size":
  ```
  size  <T>
  00  B
  01  H
  10  S
  11  D
  ```

**Operation**

```cpp
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);
V[d] = minimum<esize-1:0>;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed integer matrix multiply-accumulate

The signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

**SVE (FEAT_I8MM)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Zm | 1  | 0  | 0  | 1  | 1  | 0  | Zn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**SMMLA <Zda>.S, <Zn>.B, <Zm>.B**

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;

**Assembler Symbols**

- **<Zda>** Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1
  op1 = Elem[operand1, s, 128];
  op2 = Elem[operand2, s, 128];
  addend = Elem[operand3, s, 128];
  res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
  Elem[result, s, 128] = res;

Z[da] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMULH

Signed multiply returning high half (predicated)

Widening multiply signed integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

If \( \text{HaveSVE}() \) then UNDEFINED;

\[
\text{integer esize} = 8 \ll \text{UInt}(\text{size});
\]
\[
\text{integer } g = \text{UInt}(\text{Pg});
\]
\[
\text{integer } dn = \text{UInt}(\text{Zdn});
\]
\[
\text{integer } m = \text{UInt}(\text{Zm});
\]
\[
\text{boolean unsigned} = \text{FALSE};
\]

Assembler Symbols

\(<\text{Zdn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\(\text{CheckSVEEnabled}();\)
\(\text{integer elements} = \text{VL} \ \text{DIV} \ \text{esize};\)
\(\text{bits(PL) mask} = P[g];\)
\(\text{bits(VL) operand1} = Z[dn];\)
\(\text{bits(VL) operand2} = \text{if AnyActiveElement}(\text{mask}, \text{esize}) \text{ then } Z[m] \text{ else } \text{Zeros}();\)
\(\text{bits(VL) result};\)

for e = 0 to elements-1
\(\text{integer element1} = \text{Int(Elem[operand1, e, esize], unsigned);}\)
\(\text{integer element2} = \text{Int(Elem[operand2, e, esize], unsigned);}\)
\(\text{if Elem[mask, e, esize]} = '1' \text{ then}\)
\(\text{integer product} = (\text{element1} * \text{element2}) \gg \text{esize};\)
\(\text{Elem[result, e, esize]} = \text{product}<\text{esize-1:0}>;\)
\(\text{else}\)
\(\text{Elem[result, e, esize]} = \text{Elem[operand1, e, esize]};\)
\(Z[dn] = \text{result};\)

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SPLICE

Splice two vectors under predicate control

Copy the first active to last active elements (inclusive) from the first source vector to the lowest-numbered elements of
the result. Then set any remaining elements of the result to a copy of the lowest-numbered elements from the second
source vector. The result is placed destructively in the first source vector.

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 0 1 0 1               | size            | Pg              | Zm              |
|                               | 1 0 1 1 0 0 1 0 0 | Zdn             |                 |

SPLICE \(<Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

\(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer x = 0;
boolean active = FALSE;
integer lastnum = LastActiveElement(mask, esize);
if lastnum >= 0 then
  for e = 0 to lastnum
    active = active || ElemP[mask, e, esize] == '1';
  if active then
    Elem[result, x, esize] = Elem[operand1, e, esize];
    x = x + 1;
  elements = (elements - x) - 1;
  for e = 0 to elements
    Elem[result, x, esize] = Elem[operand2, e, esize];
    x = x + 1;
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQADD (immediate)

Signed saturating add immediate (unpredicated)

Signed saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>imm8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zdn</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
</tr>
</tbody>
</table>

SQADD <Zdn>..<T>, <Zdn>..<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQADD (vectors)

Signed saturating add vectors (unpredicated)

Signed saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & | & \text{size} & | & 1 & \text{Zm} & 0 & 0 & 0 & 1 & 0 & 0 & | & \text{Zn} & | & \text{Zd} & \\
\end{array}
\]

SQADD \(<\text{Zd}>, <\text{T}>, <\text{Zn}>, <\text{T}>, <\text{Zm}>, <\text{T}>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;

Assembler Symbols

\(<\text{Zd}>\) Is the name of the destination scalable vector register, encoded in the “Zd” field.

\(<\text{T}>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{T}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the “Zn” field.

\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = \(VL\) DIV esize;
bits(\(VL\)) operand1 = \(Z[n]\);
bits(\(VL\)) operand2 = \(Z[m]\);
bits(\(VL\)) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);
\(Z[d]\) = result;
SQDECB

Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 imm4 1 1 1 1 1 1 0 pattern Rdn</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt; sf D U</td>
</tr>
</tbody>
</table>

SQDECB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```assembly
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0 1 1 1 imm4 1 1 1 1 1 1 0 pattern Rdn</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt; sf D U</td>
</tr>
</tbody>
</table>

SQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

```assembly
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

### Assembler Symbols

- `<Xdn>`  
  Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

- `<Wdn>`  
  Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

- `<pattern>`  
  Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm>
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDECD (scalar)**

Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#).

### 32-bit

```
0 0 0 0 0 1 0 0 1 1 1 0 imm4 1 1 1 1 1 1 0 pattern Rdn
```

- `size<1>`: size<0>
- `sf`: D U

```
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

```
0 0 0 0 0 1 0 0 1 1 1 1 imm4 1 1 1 1 1 1 1 1 0 pattern Rdn
```

- `size<1>`: size<0>
- `sf`: D U

```
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

**Assembler Symbols**

* `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
* `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
* `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECD (vector)

Signed saturating decrement vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>1</td>
<td>1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

SQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQDECH (scalar)**

Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | imm4 | 1 | 1 | 1 | 1 | 1 | 0 | pattern | Rdn |
| size<1> | size<0> | sf | D | U |

**SQDECH** <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```c
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | imm4 | 1 | 1 | 1 | 1 | 1 | 0 | pattern | Rdn |
| size<1> | size<0> | sf | D | U |

**SQDECH** <Xdn>{, <pattern>{, MUL #<imm>}}

```c
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

### Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
### Pattern

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1101</td>
<td>MUL4</td>
</tr>
<tr>
<td>1110</td>
<td>MUL3</td>
</tr>
<tr>
<td>1111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

### Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECH (vector)

Signed saturating decrement vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D U size&lt;1&gt;</td>
</tr>
</tbody>
</table>

SQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}}

if !HaveSVE() then UNDEFINED;
integer size = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDECP (scalar)

Signed saturating decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & \text{Pm} & \text{Rdn} \\
\end{array}
\]

SQDECP <Xdn>, <Pm>.<T>, <Wdn>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & \text{Pm} & \text{Rdn} \\
\end{array}
\]

SQDECP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECP (vector)

Signed saturating decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

### Assembler Symbols

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pm>` Is the name of the source scalable predicate register, encoded in the "Pm" field.

### Operation

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
```

```c
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);
Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDECW (scalar)

Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 1 1 0 | pattern | Rdn
```

SQDECW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```java
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 1 1 0 | pattern | Rdn
```

SQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

```java
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11011</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12.rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECW (vector)

Signed saturating decrement vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 0</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt; D U</td>
</tr>
</tbody>
</table>

SQDECW <$Zdn>.S{, <pattern>{, MUL #<imm>}}$

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

$<Zdn>$  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

$<pattern>$  Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

$<imm>$  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating increment scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

```
0 0 0 0 0 1 0 0 | 0 0 1 0 | imm4 | 1 1 1 1 | 0 0 | pattern | Rdn

size<1> | size<0> | sf | D | U
```

**SQINCB** `<Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}`

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

```
0 0 0 0 0 1 0 0 | 0 0 1 1 | imm4 | 1 1 1 1 | 0 0 | pattern | Rdn

size<1> | size<0> | sf | D | U
```

**SQINCB** `<Xdn>{, <pattern>{, MUL #<imm>}}`

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

**Assembler Symbols**

- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQINCD (scalar)

Signed saturating increment scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

```
31 30 29 28 27 26 25 24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0 0 0 0 0 0 1 0 0         1 1 1 0 imm4 1 1 1 0 0        pattern   Rdn
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

### 64-bit

```
31 30 29 28 27 26 25 24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0 0 0 0 0 0 1 0 0         1 1 1 1 imm4 1 1 1 0 0        pattern   Rdn
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

**Assembler Symbols**

- `<Xdn>` is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<Wdn>` is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<pattern>` is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
<pattern>

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrcl, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQINCD (vector)**

Signed saturating increment vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```assembly
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

**Assembler Symbols**

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCH (scalar)

Signed saturating increment scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | imm4| 1  | 1  | 1  | 1  | 0  | 0  | pattern | Rdn |

size<1>size<0> sf D U

SQINCH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```cpp
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | imm4| 1  | 1  | 1  | 1  | 0  | 0  | pattern | Rdn |

size<1>size<0> sf D U

SQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

```cpp
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

**Assembler Symbols**

<Xdn>  Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn>  Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10011</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12 rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQINCH (vector)

Signed saturating increment vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
</table>
0 0 0 0 0 1 0 0 0 0 1 1 0 imm4 1 1 0 0 0 0 pattern | Zdn |
```

SQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCP (scalar)

Signed saturating increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar
destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated
result is then sign-extended to 64 bits.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
|    0 0 1 0 0 1 0 1     |    1 0 1 0 0 0 0 1     |    Pm               |    Rdn               |
|  D     U                 |                      |                  |

SQINCP <Xdn>, <Pm>.<T>, <Wdn>

if !HaveSVE() then UNDEFINED;
in integer esize = 8 << UInt(size);
in integer m = UInt(Pm);
in integer dn = UInt(Rdn);
boolean unsigned = FALSE;
in integer ssize = 32;

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
|    0 0 1 0 0 1 0 1     |    1 0 1 0 0 0 0 1     |    Pm               |    Rdn               |
|  D     U                 |                      |                  |

SQINCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
in integer esize = 8 << UInt(size);
in integer m = UInt(Pm);
in integer dn = UInt(Rdn);
boolean unsigned = FALSE;
in integer ssize = 64;

Assembler Symbols

<Xdn>  Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm>  Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T>  Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Wdn>  Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
SQINCP (vector)

Signed saturating increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCW (scalar)

Signed saturating increment scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 0 1 0 imm4 1 1 1 1 0 0 pattern Rdn
```

SQINCW <Xdn>, <Wdn>{, <pattern>{, MUL #imm}}

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 0 1 1 imm4 1 1 1 1 0 0 pattern Rdn
```

SQINCW <Xdn>{, <pattern>{, MUL #imm}}

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

`<Xdn>`  Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

`<Wdn>`  Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

`<pattern>`  Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```
SQINCW (vector)

Signed saturating increment vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Unspec<1>spec<0> D U

SQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQSUB (immediate)

Signed saturating subtract immediate (unpredicated)

Signed saturating subtract of an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | 1 0 0 1 1 0 1 1 | sh | imm8 | Zdn |

SQSUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned); 
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating subtract vectors (unpredicated)

Signed saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}-1)\). This instruction is unpredicated.

```
 0 0 0 0 0 1 0 0 | size | 1 | Zm 0 0 0 1 1 0 | Zn | Zd
```

**SQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>**

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);
Z[d] = result;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ST1B (scalar plus immediate)**

Contiguous store bytes from vector (immediate index)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>

**ST1B** { <Zt>.<T> }, <Pg>, [ <Xn|SP>{}, #<imm>, MUL VL }

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        src = Z[t];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
bites(64) addr = base + eoff * mbytes;
Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

ST1B (scalar plus immediate)
ST1B (scalar plus scalar)

Contiguous store bytes from vector (scalar index)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | size | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

ST1B { <Zt>, <T> }, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == "11111" then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
  src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

ST1B (scalar plus vector)

Scatter store bytes from a vector (vector index)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements are not written to memory.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

### 32-bit unpacked unscaled offset

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | Zm | 1  | xs | 0  | Pg | Rn | Zt |
| msz<1>msz<0> |
```

```plaintext
ST1B { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 1  | xs | 0  | Pg | Rn | Zt |
| msz<1>msz<0> |
```

```plaintext
ST1B { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | Zm | 1  | 0  | 1  | Pg | Rn | Zt |
| msz<1>msz<0> |
```

```plaintext
ST1B (scalar plus vector)
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrunpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
    src = Z[t];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
ST1B (vector plus immediate)

Scatter store bytes from a vector (immediate index)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

![32-bit element diagram](image)

```plaintext
ST1B { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offset = UInt(imm5);
```

### 64-bit element

![64-bit element diagram](image)

```plaintext
ST1B { <Zt>.D }, <Pg>, [<Zn].D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offset = UInt(imm5);
```

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTExe() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n];
    src = Z[t];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<mSize-1:0>;
**ST1D (scalar plus immediate)**

Contiguous store doublewords from vector (immediate index)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>
```

```
ST1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```plaintext
if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 64;
integer offset = SInt(imm4);

if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrAnUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Assembler Symbols

- `<Zt>`
- `<Pg>`
- `<Xn|SP>`
- `<imm>`

### Copyright

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1D (scalar plus scalar)

Contiguous store doublewords from vector (scalar index)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

ST1D {<Zt> .D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAinement();
    else
        if n == 31 then CheckSPAinement();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
        src = Z[t];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1D (scalar plus vector)

Scatter store doublewords from a vector (vector index)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements are not written to memory.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset.

32-bit unpacked scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 0 0 1 0 1 | 1 | 1 | 1 | 1 | 0 1 | Zm | 1 | xs | 0 | Pg | Rn | Zt |
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

32-bit unpacked unscaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 0 0 1 0 1 | 1 | 1 | 1 | 0 1 | Zm | 1 | xs | 0 | Pg | Rn | Zt |
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit scaled offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 0 0 1 0 | 1 | 1 | 1 | 0 1 | Zm | 1 | 0 1 | Pg | Rn | Zt |
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
imteger n = UInt(Rn);
imteger m = UInt(Zm);
imteger g = UInt(Pg);
imteger esize = 64;
imteger msize = 64;
imteger offs_size = 64;
boolean offs_unsigned = TRUE;
imteger scale = 3;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

Assembler Symbols

<Zt>        Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>        Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>      Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>        Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod>       Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = Z[m];
        src = Z[t];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

ST1D (vector plus immediate)

Scatter store doublewords from a vector (immediate index)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements are not written to memory.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>msz&lt;1&gt; msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

ST1D { <Zt>.D }, <Pg>, [<Zn>.D[, #<imm>]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if AnyActiveElement(mask, esize) then
    base = Z[n];
    src = Z[t];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1H (scalar plus immediate)

Contiguous store halfwords from vector (immediate index)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        src = Z[t];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
ST1H (scalar plus scalar)

Contiguous store halfwords from vector (scalar index)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | size | Rm | 0 | 1 | 0 | Pg | Rn | Zt |

ST1H \{ <Zt>,<T>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 16;

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T>  Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(P) mask = P[g];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
    src = Z[t];
  for e = 0 to elements-1
    if ElemP(mask, e, esize) == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

ST1H (scalar plus scalar)  Page 2407
ST1H (scalar plus vector)

Scatter store halfwords from a vector (vector index)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

```
msz<1>msz<0>
```

```
ST1H { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

```
msz<1>msz<0>
```

```
ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset

```
msz<1>msz<0>
```

```
ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
ST1H (scalar plus vector)

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 0 0 0 1 1 0 1 x s 0 Zm 1 Zm 0 1 Zm 1 Zm 1 0 0

64-bit scaled offset

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) mask = P[g];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = Z[m];
    src = Z[t];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[off, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1H (vector plus immediate)

Scatter store halfwords from a vector (immediate index)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements are not written to memory.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | imm5 | 1  | 0  | 1  | Pg | Zn | Zt | msz<1> | msz<0> |

**ST1H** { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offset = UInt(imm5);
```

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | imm5 | 1  | 0  | 1  | Pg | Zn | Zt | msz<1> | msz<0> |

**ST1H** { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offset = UInt(imm5);
```

### Assembler Symbols

- `<Zt>`  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`  Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
  base = Z[n];
  src = Z[t];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
**ST1W (scalar plus immediate)**

Contiguous store words from vector (immediate index)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | size | 0  | imm4 | 1  | 1  | Pg  | Rn  | Zt |

**ST1W** {<Zt>..<T>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}

if ![HaveSVE()] then UNDEFINED;
if size != '1x' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 32;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(n !!= 31);
if ![AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

ST1W (scalar plus immediate)
ST1W (scalar plus scalar)

Contiguous store words from vector (scalar index)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ST1W {<Zt>,<T>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xn> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];
        src = Z[t];
    for e = 0 to elements-1
        if ELEM[mask, e, esize] == '1' then
            bits(64) addr = base + (UInt(offset) + e) * mbytes;
            Mem[addr, mbytes, AccType_SVE] = ELEM[src, e, esize]<msize-1:0>;
ST1W (scalar plus vector)

Scatter store words from a vector (vector index)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

| 31 30 29 28 27 26 25 | 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------|-----------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 1 1 1 0 0 1 0 | 1 | 0 | 1 1 | Zm | 1 | xs | 0 | Pg | Rn | Zt |

msz<1>msz<0>

ST1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked scaled offset

| 31 30 29 28 27 26 25 | 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------|-----------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 1 1 1 0 0 1 0 | 1 | 0 | 0 1 | Zm | 1 | xs | 0 | Pg | Rn | Zt |

msz<1>msz<0>

ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked unscaled offset

| 31 30 29 28 27 26 25 | 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------|-----------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 1 1 1 0 0 1 0 | 1 | 0 | 0 0 | Zm | 1 | xs | 0 | Pg | Rn | Zt |

msz<1>msz<0>
ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>Zm 1 xs 0 Pg Rn Zt</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

ST1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>Zm 1 0 1 Pg Rn Zt</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Zm 1 0 1 Pg Rn Zt</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = Z[m];
        src = Z[t];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
ST1W (vector plus immediate)

Scatter store words from a vector (immediate index)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements are not written to memory.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

```
0 1 1 1 1 0 0 1 0 | 1 0 | 0 1 1 | imm5 | 1 0 1 | Pg | Zn | Zt
```

If !HaveSVE() then UNDEFINED;

```
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offset = UInt(imm5);
```

64-bit element

```
0 1 1 1 1 1 0 0 1 0 | 1 0 | 0 1 0 | imm5 | 1 0 1 | Pg | Zn | Zt
```

If !HaveSVE() then UNDEFINED;

```
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offset = UInt(imm5);
```

Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
  base = Z[n];
  src = Z[t];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ST2B (scalar plus immediate)**

Contiguous store two-byte structures from two vectors (immediate index)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```plaintext
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| 1 1 1 0 0 1 0 | 0 | 0 | 0 | 1 | 1 | imm4 | 1 | 1 | Pg | Rn | Zt |

**Assembler Symbols**

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

**Operation**

CheckSVE enabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31); 

if !AnyActiveElement(mask, esize) then 
  if n == 31 && ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then 
    CheckSPAlignment();
  else 
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
  
for r = 0 to nreg-1 
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1 
  for r = 0 to nreg-1 
    if ElemP[mask, e, esize] == '1' then 
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```
ST2B (scalar plus scalar)

Contiguous store two-byte structures from two vectors (scalar index)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 0 | 0 | 0 0 1 | Rm | 0 1 1 | Pg | Rn | Zt
msz<1>msz<0>
```

ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 2;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(P) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```
ST2B (scalar plus scalar)
ST2D (scalar plus immediate)

Contiguous store two-doubleword structures from two vectors (immediate index)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

msz<1>msz<0>  

ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 &
    ConstrinUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSAlignment();
    else
        if n == 31 then CheckSAlignment();
        base = if n == 31 then SP[] else X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if Elem[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
**ST2D (scalar plus scalar)**

Contiguous store two-doubleword structures from two vectors (scalar index)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```assembly
ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [ <Xn|SP>, <Xm> ], LSL #3
```

- `<Zt>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Assembler Symbols**

- `msz<1>msz<0>`
- `Rm`
- `Pg`
- `Rn`
- `Zt`
- `integer t = UInt(Zt);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `integer g = UInt(Pg);`
- `integer esize = 64;`
- `integer nreg = 2;`

If `!HaveSVE()` then UNDEFINED;
If `Rm == '11111'` then UNDEFINED;
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
ST2H (scalar plus immediate)

Contiguous store two-halfword structures from two vectors (immediate index)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

```plaintext
internal version only: isa v33.16decrel, advsimd v29.05, pseudocode v2021-12_rel, sve v2021-12

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 2;

checkSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
   if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      if n == 31 then CheckSPAlignment();
      base = if n == 31 then SP[] else X[n];
for r = 0 to nreg-1
   values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
   for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
         integer eoff = (offset * elements * nreg) + (e * nreg) + r;
         bits(64) addr = base + eoff * mbytes;
         Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```
**ST2H (scalar plus scalar)**

Contiguous store two-halfword structures from two vectors (scalar index)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

ST2H \{ <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 2;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST2W (scalar plus immediate)

Contiguous store two-word structures from two vectors (immediate index)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
msz<1>msz<0>

ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bags(64) base;
bags(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bags(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if Elem[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
bags(64) addr = base + eoff * mbytes;
Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
ST2W (scalar plus scalar)

Contiguous store two-word structures from two vectors (scalar index)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

$$
\begin{align*}
&1 1 1 0 0 1 0 \ 1 \ 0 \ 0 \ 1 \ \ Rm \ 0 \ 1 \ 1 \ \ Pg \ \ Rn \ \ Zt \\
&msz<1>msz<0>
\end{align*}
$$

ST2W \{ \ <Zt1>.S, \ <Zt2>.S \}, \ <Pg>, \ [<Xn|SP>, \ <Xm>, \ LSL \ #2] \n
if ! HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 2;
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
**ST3B (scalar plus immediate)**

Contiguous store three-byte structures from three vectors (immediate index)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    for r = 0 to nreg-1
      values[r] = Z[(t+r) MOD 32];
    for e = 0 to elements-1
      for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
          integer eoff = (offset * elements * nreg) + (e * nreg) + r;
          bits(64) addr = base + eoff * mbytes;
          Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```
ST3B (scalar plus scalar)

Contiguous store three-byte structures from three vectors (scalar index)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```plaintext
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 3;
```

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
ST3D (scalar plus immediate)

Contiguous store three-doubleword structures from three vectors (immediate index)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 1 1 0 1 imm4 1 1 1 Pg Rn Zt
```

```
msz<1>msz<0>
```

```
ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

ST3D (scalar plus immediate)
ST3D (scalar plus scalar)

Contiguous store three-doubleword structures from three vectors (scalar index)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL \text{ DIV} e\text{size};
b\text{its}(64) base;
b\text{its}([PL]) mask = P[g];
b\text{its}(64) offset;
constant integer m\text{bytes} = esize \text{ DIV} 8;
array [0..2] of b\text{its}(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, e\text{size}) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for r = 0 to nreg-1
  values[r] = Z[(t+r) \text{ MOD} 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, e\text{size}] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      b\text{its}(64) addr = base + eoff * m\text{bytes};
      Mem[addr, m\text{bytes}, AccType_SVE] = Elem[values[r], e, e\text{size}];
ST3H (scalar plus immediate)

Contiguous store three-halfword structures from three vectors (immediate index)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} & \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{msz}<1> & \text{msz}<0> & \\
\end{array}
\]

\[
\text{ST3H \{ <Zt1>.H, <Zt2>.H, <Zt3>.H \}, <Pg>, [<Xn|SP>\{, #<imm>, MUL VL}\}}
\]

if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();

integer elements = VL DIV esize;

bits(64) base;

bits(PL) mask = P[g];

constant integer mbytes = esize DIV 8;

array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 & & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
**ST3H (scalar plus scalar)**

Contiguous store three-halfword structures from three vectors (scalar index)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
ST3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>, < Xm>, LSL #1]  
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 3;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```
ST3W (scalar plus immediate)

Contiguous store three-word structures from three vectors (immediate index)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

The assembly instruction for ST3W is:

```
```

where:

- `<Zt1>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

### Assembler Symbols

- `<Zt1>`
- `<Zt2>`
- `<Zt3>`
- `<Pg>`
- `<Xn|SP>`
- `<imm>`

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 &
      ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

ST3W (scalar plus immediate)
ST3W (scalar plus scalar)

Contiguous store three-word structures from three vectors (scalar index)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  1 1 1 0 0 1 0 1 0 Rm 0 1 1 Pg Rn Zt

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 3;
```

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = \text{VL} \div \text{esize};
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = \text{esize} \div 8;
array [0..2] of bits(\text{VL}) values;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if \text{AnyActiveElement}(mask, \text{esize}) then
  if \text{n} == 31 && \text{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) then
    CheckSPAlignment();
  else
    if \text{n} == 31 then \text{CheckSPAlignment}();
    base = if \text{n} == 31 then \text{SP[]} else \text{X}[n];
    offset = \text{X}[m];

for \text{r} = 0 to \text{nreg}-1
  values[r] = \text{Z}[\text{(t+r)} \mod 32];

for \text{e} = 0 to \text{elements}-1
  for \text{r} = 0 to \text{nreg}-1
    if \text{ElemP[mask, e, esize]} == '1' then
      integer eoff = \text{UInt}(offset) + (e * \text{nreg}) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = \text{Elem}[values[r], e, esize];
ST4B (scalar plus immediate)

Contiguous store four-byte structures from four vectors (immediate index)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 4;
```

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem(values[r], e, esize);
**ST4B (scalar plus scalar)**

Contiguous store four-byte structures from four vectors (scalar index)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

![Address Calculation](attachment:image.png)

```plaintext
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 4;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];
for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```
ST4D (scalar plus immediate)

Contiguous store four-doubleword structures from four vectors (immediate index)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>

msz<1>msz<0>


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```cpp
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = (offset * elements * nreg) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ST4D (scalar plus scalar)**

Contiguous store four-doubleword structures from four vectors (scalar index)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
ST4D (scalar plus scalar)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1
msz<1> msz<0>

ST4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if ! HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 4;
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
**ST4H (scalar plus immediate)**

Contiguous store four-halfword structures from four vectors (immediate index)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
------|----|----|----|----|----|----|----|----|----|----|----|------|----|----|----|----|----|----|
msz<1>msz<0>
```


```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 4;
```

### Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ELEM[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
ST4H (scalar plus scalar)

Contiguous store four-halfword structures from four vectors (scalar index)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 1 1 0 0 1 0 | 0 | 1 | 1 | Rm | 0 | 1 | 1 | Pg | Rn | Zt

msz<1>msz<0>


if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        offset = X[m];

    for r = 0 to nreg-1
        values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = Uint(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST4W (scalar plus immediate)

Contiguous store four-word structures from four vectors (immediate index)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.

$\text{ST4W} \{ \text{<Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, \text{ <Pg}, \text{ [<Xn|SP}{, #<imm>, MUL VL} \}

\text{if !HaveSVE()} \text{ then UNDEFINED;}
integer t = \text{ UInt}(Zt);
integer n = \text{ UInt}(Rn);
integer g = \text{ UInt}(Pg);
integer esize = 32;
integer offset = \text{ SInt}(\text{imm4});
integer nreg = 4;

Assembler Symbols

$\text{<Zt1>}$ Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
$\text{<Zt2>}$ Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
$\text{<Zt3>}$ Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
$\text{<Zt4>}$ Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
$\text{<Pg>}$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
$\text{<Xn|SP>}$ Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
$\text{<imm>}$ Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
ST4W (scalar plus scalar)

Contiguous store four-word structures from four vectors (scalar index)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 1 1 0 0 1 0 | 1 0 1 1 | Rm 0 1 1 | Pg | Rn | Zt |

msz<1>msz<0>

ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    offset = X[m];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      integer eoff = UInt(offset) + (e * nreg) + r;
      bits(64) addr = base + eoff * mbytes;
      Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
STNT1B (scalar plus immediate)

Contiguous store non-temporal bytes from vector (immediate index)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
+-------+-------+-------+-------+-------+-------+-------+-------+
| 31    | 30    | 29    | 28    | 27    | 26    | 25    | 24    | 23    | 22    | 21    | 20    | 19    | 18    | 17    | 16    | 15    | 14    | 13    | 12    | 11    | 10    | 9     | 8     | 7     | 6     | 5     | 4     | 3     | 2     | 1     |
| 1     | 1     | 1     | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     | imm4  | 1     | 1     | 1     | Pg    | Rn    | Zt    |
| msz<1> | msz<0>|
```

STNT1B { <Zt> .B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSAPalignment();
else
    if n == 31 then CheckSAPalignment();
    base = if n == 31 then SP[] else X[n];
    src = Z[t];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements) + e;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```
STNT1B (scalar plus scalar)

Contiguous store non-temporal bytes from vector (scalar index)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>

msz<1>msz<0>

STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 &
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
  src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
STNT1D (scalar plus immediate)

Contiguous store non-temporal doublewords from vector (immediate index)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 1 0 0 1 0 1 1 0 0 1 imm4 1 1 1 Pg Rn Zt
```

STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #imm}, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 & & ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n];
    src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1D (scalar plus scalar)

Contiguous store non-temporal doublewords from vector (scalar index)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 1 0 0 | Rm | 0 1 1 | Pg | Rn | Zt
   msz<1> msz<0>

STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 &
  ConstrUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
  src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

STNT1H (scalar plus immediate)

Contiguous store non-temporal halfwords from vector (immediate index)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 1 0 0 1 0 0 | 0 | 1 | 0 | 0 | 1 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |
msz<1>msz<0>

STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if !HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        if n == 31 then CheckSPAlignment();
        base = if n == 31 then SP[] else X[n];
        src = Z[t];
    for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

STNT1H (scalar plus scalar)

Contiguous store non-temporal halfwords from vector (scalar index)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 1 1 1 0 0 1 0 | 0 | 1 | 0 | 0 | Rm | 0 | 1 | 1 | Pg | 0 | 0 | Rn | 0 | Zt | 0 | msz<1> | msz<0> |

STNT1H { <Zt> .H }, <Pg>, [<Xn|SP>], <Xm>, LSL #1

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

Assembler Symbols

<Zt>     Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>     Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>     Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
binary(64) base;
binary(64) offset;
binary(VL) src;
binary(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
  src = Z[t];
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
      bits(64) addr = base + (UInt(offset) + e) * mbytes;
      Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
STNT1W (scalar plus immediate)

Contiguous store non-temporal words from vector (immediate index)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 1 1 0 0 1 0 | 1 | 0 | 0 | 0 | imm4 | 1 | 1 | Pg | Rn | Zt |
```

STNT1W { <Zt>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);

Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if HaveMTEExt() then SetTagCheckedInstruction(n != 31);
if !AnyActiveElement(mask, esize) then
  if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer eoff = (offset * elements) + e;
    bits(64) addr = base + eoff * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1W (scalar plus scalar)

Contiguous store non-temporal words from vector (scalar index)

Contiguous store non-temporal words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | Rm | 0 | 1 | 1 | Pg | Rn | Zt |

msz<1>msz<0>

STNT1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if !AnyActiveElement(mask, esize) then
  if n == 31 & ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then CheckSPAlignment();
else
  if n == 31 then CheckSPAlignment();
  base = if n == 31 then SP[] else X[n];
  offset = X[m];
  src = Z[t];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(64) addr = base + (UInt(offset) + e) * mbytes;
    Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
STR (predicate)

Store predicate register

Store a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The store is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 1 0</td>
</tr>
</tbody>
</table>

STR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler Symbols

<Pt> Is the name of the scalable predicate transfer register, encoded in the "Pt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
integer elements = PL DIV 8;
bits(PL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
  CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

src = P[t];
boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_SVE, TRUE);
for e = 0 to elements-1
  AArch64_MemSingle(base + offset, 1, AccType_SVE, aligned) = Elem[src, e, 8];
  offset = offset + 1;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STR (vector)

Store vector register

Store a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 1 0</td>
</tr>
</tbody>
</table>

STR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV 8;
bits(VL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

src = Z[t];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SVE, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned] = Elem[src, e, 8];
    offset = offset + 1;
**SUB (immediate)**

Subtract immediate (unpredicated)

Subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

```
| 0 0 1 0 0 1 0 1 | size | 1 0 0 0 0 1 1 1 | sh | imm8 | Zdn |
```

**Assembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the “imm8” field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```
if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = element1 - imm;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SUB (vectors, predicated)**

Subtract vectors (predicated)

Subtract active elements of the second source vector from corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
```

**SUB <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>**

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size | <T>
  - 00  | B
  - 01  | H
  - 10  | S
  - 11  | D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SUB (vectors, unpredicated)

Subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = element1 - element2;
Z[d] = result;
```
**SUBR (immediate)**

Reversed subtract from immediate (unpredicated)

Reversed subtract from an unsigned immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | 1 0 0 | 0 1 1 1 | sh | imm8 | Zdn |
---|---|---|---|---|---|---|

**SUBR <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}**

```plaintext
if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

**Assembler Symbols**

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>` is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (imm - element1)<esize-1:0>;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SUBR (vectors)

Reversed subtract vectors (predicated)

Reversed subtract active elements of the first source vector from corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | size | 0 0 0 | 0 1 1 0 0 0 | Pg | Zm | Zdn
```

```
SUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element2 - element1;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed by unsigned integer indexed dot product

The signed by unsigned integer indexed dot product instruction computes the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated. ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

**Assembler Symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm> Is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = Uint(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SUNPKHI, SUNPKLO

Signed unpack and extend half of vector

Unpack elements from the lowest or highest half of the source vector and then sign-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: **High half** and **Low half**

**High half**

![Binary representation of high half encoding](image)

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer n = Uint(Zn);
integer d = Uint(Zd);
boolean unsigned = FALSE;
boolean hi = TRUE;
```

**Low half**

![Binary representation of low half encoding](image)

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer n = Uint(Zn);
integer d = Uint(Zd);
boolean unsigned = FALSE;
boolean hi = FALSE;
```

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
integer hsize = \texttt{esize} \texttt{DIV} 2;
bits(\texttt{VL}) operand = \texttt{Z}[n];
bits(\texttt{VL}) result;

for e = 0 to elements-1
    \texttt{bits(hsize)} element = if hi then \texttt{Elem}[operand, e + elements, hsize] else \texttt{Elem}[operand, e, hsize];
    \texttt{Elem}[result, e, esize] = \texttt{Extend}(element, esize, unsigned);
\texttt{Z}[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTB, SXTH, SXTW

Signed byte / halfword / word extend (predicated)

Sign-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: Byte, Halfword and Word

**Byte**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

**Halfword**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

**Word**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SXTW <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

For the halfword variant: is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);
Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Programmable table lookup in single vector table

Reads each element of the second source (index) vector and uses its value to select an indexed element from the first source (table) vector, and places the indexed table element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then it places zero in the corresponding destination vector element.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

TBL <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:
  ```plaintext
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer idx = UInt(Elem[operand2, e, esize]);
  Elem[result, e, esize] = if idx < elements then Elem[operand1, idx, esize] else Zeros();
Z[d] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
TRN1, TRN2 (predicates)

Interleave even or odd elements from two predicates

Interleave alternating even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: Even and Odd

Even

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pn</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

H

TRN1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;

Odd

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pn</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

H

TRN2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

```c
CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];

P[d] = result;
```

TRN1, TRN2 (vectors)

Interleave even or odd elements from two vectors

Interleave alternating even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented. It has encodings from 4 classes: Even, Even (quadwords), Odd and Odd (quadwords)

Even

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  0  1  0  1  1  0  0  0  0  1  0  1  1  0  0  1  1  0  0
  Zm  Zn  Zd
  
      H
```

TRN1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Even (quadwords)

(FEAT_F64MM)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  0  1  0  1  1  0  1  0  0  0  1  0  1  0  1  0  1  0
  Zm  Zn  Zd
  
      H
```

TRN1 <Zd>.Q, <Zn>.Q, <Zm>.Q

```java
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Odd

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  0  1  0  1  1  0  1  0  0  0  1  0  1  1  0  1  1  0  0
  Zm  Zn  Zd
  
      H
```

TRN2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```
Odd (quadwords)
(_FEAT_F64MM)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

\[ TRN2 \langle Zd \rangle.Q, \langle Zn \rangle.Q, \langle Zm \rangle.Q \]

if \(!\)HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\(\text{CheckSVEEnabled}()\);
if \(\text{VL} \lt \text{esize} \times 2\) then UNDEFINED;
integer pairs = \(\text{VL} \div \text{esize} \times 2\);
bits(\text{VL}) operand1 = \(Z[n]\);
bits(\text{VL}) operand2 = \(Z[m]\);
bits(\text{VL}) result = \(\text{Zeros}()\);
for \(p = 0\) to \(\text{pairs-1}\)
\(\text{Elem}[\text{result}, 2*p+0, \text{esize}] = \text{Elem}[\text{operand1}, 2*p+\text{part}, \text{esize}]\);
\(\text{Elem}[\text{result}, 2*p+1, \text{esize}] = \text{Elem}[\text{operand2}, 2*p+\text{part}, \text{esize}]\);
\(Z[d] = \text{result}\);
UABD

Unsigned absolute difference (predicated)

Compute the absolute difference between unsigned integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
unsigned abs diff (predicated)
```

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|---------------------|---------------------|
| 0 0 0 0 0 0 1 0 0 | size | 0 0 1 | 1 0 1 0 0 0 | Pg | Zm | Zdn |

UABD \<Zdn>., \<Pg>/M, \<Zdn>.<T>, \<Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

- \<Zdn>\ Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- \<T>\ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- \<Pg>\ Is the name of the governing scalable predicate register P0-P7, encoded in the " Pg" field.
- \<Zm>\ Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer absdiff = Abs(element1 - element2);
    Elem[result, e, esize] = absdiff<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UADDV

Unsigned add reduction to scalar

Unsigned add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 | Pg | Zn | Vd |

UADDV <Dd>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
integer sum = 0;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand, e, esize]);
        sum = sum + element;

V[d] = sum<63:0>;

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned integer convert to floating-point (predicated)

Convert to floating-point from the unsigned integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: 16-bit to half-precision, 32-bit to half-precision, 32-bit to single-precision, 32-bit to double-precision, 64-bit to half-precision, 64-bit to single-precision and 64-bit to double-precision

16-bit to half-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

int_U
```

UCVTF <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

32-bit to half-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

int_U
```

UCVTF <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

32-bit to single-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

int_U
```
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------|---|---|---|
| 0 1 1 0 0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 | Pg | Zn | Zd |

UCVTF <Zd>.D, <Pg>/M, <Zn>.S

64-bit to half-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------|---|---|---|
| 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 1 0 1 | Pg | Zn | Zd |

UCVTF <Zd>.H, <Pg>/M, <Zn>.D

64-bit to single-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------|---|---|---|
| 0 1 1 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 | Pg | Zn | Zd |
UCVTF &lt;Zd&gt;.S, &lt;Pg&gt;/M, &lt;Zn&gt;.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

**64-bit to double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  |

int_U

UCVTF &lt;Zd&gt;.D, &lt;Pg&gt;/M, &lt;Zn&gt;.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

**Assembler Symbols**

&lt;Zd&gt; Is the name of the destination scalable vector register, encoded in the "Zd" field.
&lt;Pg&gt; Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
&lt;Zn&gt; Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

CheckSVEenabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];

for e = 0 to elements-1
  if Elem(mask, e, esize) == '1' then
    bits(esize) element = Elem(operand, e, esize);
    bits(d_esize) fpval = FixedToFP(element&lt;s_esize-1:0&gt;, 0, unsigned, FPCR[], rounding);
    Elem(result, e, esize) = ZeroExtend(fpval);

Z[d] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UDIV**

Unsigned divide (predicated)

Unsigned divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| R | U |
```

**UDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>**

```
if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

**Assembler Symbols**

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size<0>”:
  - `size<0>`<T>
    - 0: S
    - 1: D
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element2 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDIVR

Unsigned reversed divide (predicated)

Unsigned reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer quotient;
    if element1 == 0 then
      quotient = 0;
    else
      quotient = RoundTowardsZero(Real(element2) / Real(element1));
    Elem[result, e, esize] = quotient<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDOT (indexed)

Unsigned integer indexed dot product

The unsigned integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated. It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|------------------|------------------|------------------|
| 0 1 0 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 i2 Zm 0 0 0 0 0 1 Zn Zda U size<1>*size<0> |


if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|------------------|------------------|------------------|
| 0 1 0 0 0 1 0 0 1 0 1 1 1 i1 Zm 0 0 0 0 0 1 Zn Zda U size<1>*size<0> |


if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 32-bit variant: is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.
For the 64-bit variant: is the immediate index of a quadtuplet of four 16-bit elements within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.
**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDOT (vectors)

Unsigned integer dot product

The unsigned integer dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. This instruction is unpredicated.

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMAX (immediate)**

**Unsigned maximum with immediate (unpredicated)**

Determine the unsigned maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 0 1 1 1 0</td>
</tr>
</tbody>
</table>

**UMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>**

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);
```

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMAX (vectors)

Unsigned maximum vectors (predicated)

Determine the unsigned maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | Pg | Zm | Zdn |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
The MOVPRFX instruction must specify the same destination register as this instruction.

The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMAXV

Unsigned maximum reduction to scalar

Unsigned maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-----------------|-----------------|-----------------|
| 0 0 0 0 1 0 0 | size | 0 0 1 0 0 1 | Pg | Zn | Vd |

UMAXV <V><d>, <Pg>, <Zn>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size &lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B</td>
</tr>
<tr>
<td>01 H</td>
</tr>
<tr>
<td>10 S</td>
</tr>
<tr>
<td>11 D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B</td>
</tr>
<tr>
<td>01 H</td>
</tr>
<tr>
<td>10 S</td>
</tr>
<tr>
<td>11 D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
integer maximum = if unsigned then 0 else -(2^(esize-1));
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        maximum = Max(maximum, element);
V[d] = maximum<esize-1:0>;
**UMIN (immediate)**

Unsigned minimum with immediate (unpredicated)

Determine the unsigned minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|-----------------|-----------------|-----------------|
| 0                      | 0               | 0               | 0               | 0               |
| 0                      | 1               | 0               | 1               | 0               |
| 0                      | 1               | 0               | 1               | 1               |
| 0                      | 1               | 1               | 1               | 0               |
| size                   | imm8            | Zdn             |
```

**Assembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size `<T>`
    - 00 B
    - 01 H
    - 10 S
    - 11 D
- `<imm>` Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

**Operation**

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);

integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMIN (vectors)**

Unsigned minimum vectors (predicated)

Determine the unsigned minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

\[
\text{CheckSVEEnabled}();
\text{integer elements} = \text{VL} \div \text{esize};
\text{bits(PL) mask} = \text{P}[g];
\text{bits(VL) operand1} = \text{Z}[dn];
\text{bits(VL) operand2} = \begin{cases} \text{AnyActiveElement}(\text{mask}, \text{esize}) \text{ then } \text{Z}[m] \text{ else } \text{Zeros}(); \\ \text{bits(VL) result} \end{cases};
\text{for } e = 0 \text{ to elements-1}
\quad \text{integer element1} = \text{Int(Elem}[\text{operand1}, e, \text{esize}], \text{unsigned});
\quad \text{integer element2} = \text{Int(Elem}[\text{operand2}, e, \text{esize}], \text{unsigned});
\quad \text{if } \text{Elem}[\text{mask}, e, \text{esize}] \text{ == '1' then}
\quad \quad \text{integer minimum} = \text{Min(element1, element2)};
\quad \quad \text{Elem}[\text{result}, e, \text{esize}] = \text{minimum}<\text{esize-1:0}>;
\quad \text{else}
\quad \quad \text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand1}, e, \text{esize}];
\]

\[
\text{Z}[dn] = \text{result};
\]

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMINV**

Unsigned minimum reduction to scalar

Unsigned minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

**UMINV <V><d>, <Pg>, <Zn>.<T>**

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in "size":
  ```
  size <V>
  00 B
  01 H
  10 S
  11 D
  ```
- `<d>` Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in "size":
  ```
  size <T>
  00 B
  01 H
  10 S
  11 D
  ```

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = Int(Elem[operand, e, esize], unsigned);
    minimum = Min(minimum, element);

V[d] = minimum<esize-1:0>;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMMLA

Unsigned integer matrix multiply-accumulate

The unsigned integer matrix multiply-accumulate instruction multiplies the $2 \times 8$ matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the $8 \times 2$ matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting $2 \times 2$ widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE

(Feat_I8MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | Zm | 1  | 0  | 0  | 1  | 1  | 0  | Zn | Zda |

UNMLLA <Zda>, <Zn>.B, <Zm>.B

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1
    op1 = Elem(operand1, s, 128);
    op2 = Elem(operand2, s, 128);
    addend = Elem(operand3, s, 128);
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
Elem(result, s, 128) = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMULH**

Unsigned multiply returning high half (predicated)

Widening multiply unsigned integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 1 0 0 1 1 0 0</td>
<td>H</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m] else Zeros();
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer product = (element1 * element2) >> esize;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The **MOVPRFX** instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQADD (immediate)

Unsigned saturating add immediate (unpredicated)

Unsigned saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \((2^N)-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 1 0 1 1 1 sh</th>
<th>imm8</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>00 01 10 11</td>
<td>imm8</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>

UQADD <Zdn>,<T>, <Zdn>,<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();

integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1

integer element1 = Int(Elem[operand1, e, esize], unsigned);
(Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);

Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQADD (vectors)

Unsigned saturating add vectors (unpredicated)

Unsigned saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.

```
 0 0 0 0 0 1 0 0  |  size 1  |
 0 0 0 1 0 1 0 1  |  Zm    |
 0 0 1 0 1 1 0 1  |  Zn    |
 0 1 1 0 1 0 1 0  |  Zd    |
```

UQADD <Zd><T>, <Zn><T>, <Zm><T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);
Z[d] = result;
Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

<table>
<thead>
<tr>
<th>pattern</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
</tr>
</tbody>
</table>

{<Wdn>, <pattern>{, MUL #<imm>}}

```java
if (!HaveSVE()) then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

<table>
<thead>
<tr>
<th>pattern</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
</tr>
</tbody>
</table>

{<Xdn>, <pattern>{, MUL #<imm>}}

```java
if (!HaveSVE()) then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

#### Assembler Symbols

- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

UQDECB
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECD (scalar)

Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | imm4| 1  | 1  | 1  | 1  | 1  | pattern | Rdn |
| size<1> | size<0> | sf | D | U |

UQDECD <Wdn>{, <pattern>{, MUL #<imm>}}

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

### 64-bit

| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | imm4| 1  | 1  | 1  | 1  | 1  | pattern | Rdn |
| size<1> | size<0> | sf | D | U |

UQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

### Assembler Symbols

- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:  
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12.rel, sve v2021-12; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECD (vector)

Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Unsaturated <pattern> encodings generate an empty predicate.

Unsaturated <imm> encodings generate a zero multiplier.

Unspecified or out of range size encodings generate an empty predicate.

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01101</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10x01</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10x10</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQDECH (scalar)**

Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size&lt;1&gt;size&lt;0&gt;</th>
<th>sf</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

UQDECH <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size&lt;1&gt;size&lt;0&gt;</th>
<th>sf</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

UQDECH <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

**Assembler Symbols**

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11x00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1101</td>
<td>MUL4</td>
</tr>
<tr>
<td>1110</td>
<td>MUL3</td>
</tr>
<tr>
<td>1111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result = 0;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQDECH (vector)**

Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 0 1 0 1 0 imm4</td>
<td>1 1 0 0 1 1 pattern</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>

**UQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}**

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECP (scalar)

Unsigned saturating decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
0 0 1 0 0 1 0 1 | size 1 0 1 0 | 1 1 1 0 0 1 | 0 0 | Pm | Rdn
```

UQDECP <Wdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

```
0 0 1 0 0 1 0 1 | size 1 0 1 0 | 1 1 1 0 0 1 | 1 0 | Pm | Rdn
```

UQDECP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in "size":

```
size <T>
00 B
01 H
10 S
11 D
```
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decr, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECP (vector)

Unsigned saturating decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 0 1 0 0 1 0 1 | size | 1 0 1 0 1 1 1 0 0 0 0 0 | Pm | Zdn |

UQDECP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], '-') = SatQ(element - count, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQDECW (scalar)

Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 1 | 0 1 0 | imm4 | 1 | 1 | 1 | 1 | 1 | pattern | Rdn |
| size<1>size<0> | sf | D | U |

**UQDECW <Wdn>{, <pattern>{, MUL #<imm>}}**

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

### 64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 1 | 0 1 1 | imm4 | 1 | 1 | 1 | 1 | 1 | pattern | Rdn |
| size<1>size<0> | sf | D | U |

**UQDECW <Xdn>{, <pattern>{, MUL #<imm>}}**

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

**Assembler Symbols**

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECW (vector)

Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
  * A fixed number (VL1 to VL256)
  * The largest power of two (POW2)
  * The largest multiple of three or four (MUL3 or MUL4)
  * All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Uns <Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>100x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: **32-bit** and **64-bit**

**32-bit**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |

| size<1> | size<0> | sf | D | U |

**UQINCB <Wdn>{, <pattern>{, MUL #<imm>}}**

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

**64-bit**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

| size<1> | size<0> | sf | D | U |

**UQINCB <Xdn>{, <pattern>{, MUL #<imm>}}**

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

**Assembler Symbols**

- **<Wdn>** Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- **<Xdn>** Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
- **<pattern>** Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 

---

**UQINCB**

Page 2549
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>VL512</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCD (scalar)

Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is
saturated to the general-purpose register's unsigned integer range.
The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).
Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| size<1>|size<0>| sf | D | U |

UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}</p>

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| size<1>|size<0>| sf | D | U |

UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}</p>

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCD (vector)

Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size&lt;1&gt;size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1 1 1 0</td>
<td>imm4</td>
<td>1 1 0 0 0</td>
<td>1</td>
</tr>
</tbody>
</table>

UQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCH (scalar)

Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rdn|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

UQINCH <Wdn>{, <pattern>{, MUL #<imm>}}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rdn|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

UQINCH <Xdn>{, <pattern>{, MUL #<imm>}}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11101</td>
<td>MUL3</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCH (vector)

Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-------------------------|-------------------------|-------------------------|-------------------------|
| 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 0 0 0 1 | pattern | Zdn |

size<1>size<0>  D  U

UQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINCP (scalar)

Unsigned saturating increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
0 0 1 0 0 1 0 1 | size 1 0 1 0 0 1 0 0 | Pm | Rdn
D U sf
```

UQINCP <Wdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer m = Uint(Pm);
integer dn = Uint(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

```
0 0 1 0 0 1 0 1 | size 1 0 1 0 0 1 0 1 | Pm | Rdn
D U sf
```

UQINCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer m = Uint(Pm);
integer dn = Uint(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCP (vector)

Unsigned saturating increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

UQINCP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINCW (scalar)

Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | | pattern | Rdn |

size<1> size<0> sf D U

UQINCW <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | | pattern | Rdn |

size<1> size<0> sf D U

UQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

UQINCW (scalar)
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1000x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1010x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1100x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>MUL4</td>
</tr>
<tr>
<td>11100</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCW (vector)

Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an
immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The
results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 1 0 1 1 0 1 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 0 1 0</td>
</tr>
<tr>
<td>size&lt;1&gt; size&lt;0&gt; imm4 pattern Zdn D U</td>
</tr>
</tbody>
</table>

UQINCW <Zdn>.S{}, <pattern>{, MUL #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx0x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x00x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSUB (immediate)

Unsigned saturating subtract immediate (unpredicated)

Unsigned saturating subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \((2^N) - 1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
<th>&lt;Zdn&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

UQSUB: `<Zdn>.<T>', `<Zdn>.<T>', #<imm>{, <shift>}

```plaintext
if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(uimm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;
```

Assembler Symbols

- `<Zdn>`: Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>`: Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>`: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);
Z[dn] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQSUB (vectors)

Unsigned saturating subtract vectors (unpredicated)

Unsigned saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \(2^N\)-1. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

UQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);
Z[d] = result;
USDOT (indexed)

Unsigned by signed integer indexed dot product

The unsigned by signed integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector. The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated. ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE (FEAT_I8MM)

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer esize = 32;
integer index = UINT(i2);
integer n = UINT(Zn);
integer m = UINT(Zm);
integer da = UINT(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
<imm> Is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = [Z[n];
bits(VL) operand2 = [Z[m];
bits(VL) operand3 = [Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UINT(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SINT(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
        Elem[result, e, esize] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USDOT (vectors)

Unsigned by signed integer dot product

The unsigned by signed integer dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(Feat_I8MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | Zm | 0  | 1  | 1  | 1  | 1  | 0  | Zn | Zda |

size<1>|size<0>


if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;

integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
  Elem[result, e, esize] = res;

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USMMLA

Unsigned by signed integer matrix multiply-accumulate

The unsigned by signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(FEAT_I8MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | Zm | 1  | 0  | 0  | 1  | 1  | 0  | Zn | 1  | Zda |


if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UUNPKHI, UUNPKLO

Unsigned unpack and extend half of vector

Unpack elements from the lowest or highest half of the source vector and then zero-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: High half and Low half

High half

```plaintext
<p>| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|---------------------|</p>
<table>
<thead>
<tr>
<th>0 0 0 0 0 1 0 1 size 1 1 0 0 1 1 0 0 1 1 1 0</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>
```

UUNPKHI <Zd>.<T>, <Zn>.<Tb>

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = TRUE;
```

Low half

```plaintext
<p>| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|---------------------|</p>
<table>
<thead>
<tr>
<th>0 0 0 0 0 1 0 1 size 1 1 0 0 1 0 0 0 1 1 1 0</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>
```

UUNPKLO <Zd>.<T>, <Zn>.<Tb>

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer hsize = esize DIV 2;
bits(VL) operand = Z[n];
bits(VL) result;

for e = 0 to elements-1
  bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
  Elem[result, e, esize] = Extend(element, esize, unsigned);
Z[d] = result;

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UXTB, UXTH, UXTW

Unsigned byte / halfword / word extend (predicated)

Zero-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: Byte, Halfword and Word

### Byte

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| size 0 1 0 0 1 1 1 0 1 Pg Zn Zd |

UXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

### Halfword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| size 0 1 0 0 1 1 1 0 1 Pg Zn Zd |

UXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

### Word

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| size 0 1 0 1 1 0 1 1 0 1 Pg Zn Zd |

UXTW <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

For the halfword variant: is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n] else Zeros();
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);
Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UZP1, UZP2 (predicates)

Concatenate even or odd elements from two predicates

Concatenate adjacent even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: Even and Odd

**Even**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | Pm | 0 | 1 | 0 | 0 | 1 | 0 | 0 | Pn | 0 | Pd | H |

**UZP1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;

**Odd**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | Pm | 0 | 1 | 0 | 1 | 1 | 0 | Pn | 0 | Pd | H |

**UZP2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;

**Assembler Symbols**

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the “Pd” field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- **<Pn>** Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- **<Pm>** Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

```c
CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for p = 0 to pairs - 1
    Elem[result, p, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];

P[d] = result;
```
UZP1, UZP2 (vectors)

Concatenate even or odd elements from two vectors

Concatenate adjacent even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64FPSCR_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented. It has encodings from 4 classes: Even, Even (quadwords), Odd and Odd (quadwords)

Even

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 size 1 Zm 0 1 1 0 1 Zn Zd

UZP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Even (quadwords)

(FEAT_F64MM)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 1 0 1 size 1 Zm 0 0 0 0 1 0 Zn Zd

UZP1 <Zd>.Q, <Zn>.Q, <Zm>.Q

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Odd

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 size 1 Zm 0 1 1 0 1 Zn Zd

UZP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
if vl < esize * 2 then UNDEFINED;
integer pairs = vl DIV (esize * 2);
bits(vl) operand1 = Z[n];
bits(vl) operand2 = Z[m];
bits(vl) result = Zeros();

for p = 0 to pairs - 1
    Elem[result, p, esize] = Elem[operand1, 2*p+part, esize];
for p = 0 to pairs - 1
    Elem[result, pairs+p, esize] = Elem[operand2, 2*p+part, esize];
Z[d] = result;
WHILELE

While incrementing signed scalar less than or equal to scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size 1 | Rm 0 0 0 sf 0 1 | Rn 1 | Pd
```

WHILELE <Pd>.<T>, <R><n>, <R><m>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_LE;
```

Assembler Symbols

- `<Pd>`: Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<R>`: Is a width specifier, encoded in “sf”:

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

- `<n>`: Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.
- `<m>`: Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;
for e = 0 to elements-1
  boolean cond;
  case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 + 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
WHILELO

While incrementing unsigned scalar lower than scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first,
unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered
element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is
incremented by one for each destination predicate element, irrespective of the predicate result element size. The first
general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition
flags based on the predicate result, and the V flag to zero.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size | 1 | Rm | 0 0 0 sf | 1 | 1 | Rn | 0 | Pd
U lt eq

WHILELO <Pd>.<T>, <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LT;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “sf”:

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(\texttt{PL}) mask = \texttt{Ones(PL)};
bits(\texttt{rsiz}) operand1 = \texttt{X}[n];
bits(\texttt{rsiz}) operand2 = \texttt{X}[m];
bits(\texttt{PL}) result;
boolean last = \texttt{TRUE};

for \texttt{e} = 0 to elements-1
  boolean cond;
  case \texttt{op} of
    when \texttt{Cmp_LT} \texttt{cond} = (\texttt{Int}(operand1, unsigned) < \texttt{Int}(operand2, unsigned));
    when \texttt{Cmp_LE} \texttt{cond} = (\texttt{Int}(operand1, unsigned) \texttt{<=} \texttt{Int}(operand2, unsigned));
      last = last \&\& \texttt{cond};
  \texttt{ElemP}[\texttt{result, e, esize}] = if last then '1' else '0';
  operand1 = operand1 + 1;
  \texttt{PSTATE.<N,Z,C,V>} = \texttt{PredTest}(mask, result, esize);
  \texttt{P[d]} = result;
WHILELS

While incrementing unsigned scalar lower or same as scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

$$\begin{array}{cccccccccc}
\text{size} & 1 & \text{Rm} & 0 & 0 & 0 & 1 & 0 & 1 & \text{Pd}
\end{array}$$

WHILELS <Pd>.<T>, <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in "sf":

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

CheckSVEEnabled();
iinteger elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
  boolean cond;
  case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
  
    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
WHILELT

While incrementing signed scalar less than scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element. The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

\[
\begin{array}{cccccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & Rm & 0 & 0 & 0 & sf & 0 & 1 & Rn & 0 & Pd & U & lt & eq \\
\end{array}
\]

WHILELT <Pd>,<T>, <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_LT;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “sf”:

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
operand1 = operand1 + 1;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
WRFFR

Write the first-fault register

Read the source predicate register and place in the first-fault register (FFR). This instruction is intended to restore a saved FFR and is not recommended for general use by applications.

This instruction requires that the source predicate contains a MONOTONIC predicate value, in which starting from bit 0 there are zero or more 1 bits, followed only by 0 bits in any remaining bit positions. If the source is not a monotonic predicate value, then the resulting value in the FFR will be UNPREDICTABLE. It is not possible to generate a non-monotonic value in FFR when using SETFFR followed by first-fault or non-fault loads.

```
0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0 0 Pn 0 0 0 0 0
```

if !HaveSVE() then UNDEFINED;
integer n = UInt(Pn);

Asmbleer Symbols

<\Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
bits(PL) operand = P[n];

hsb = HighestSetBit(operand);
if hsb < 0 || IsOnes(operand<hsb:0>) then
  FFR[] = operand;
else // not a monotonic predicate
  FFR[] = bits(PL) UNKNOWN;
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ZIP1, ZIP2 (predicates)

Interleave elements from two half predicates

Interleave alternating elements from the lowest or highest halves of the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.
It has encodings from 2 classes: *High halves* and *Low halves*

**High halves**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |   | Pm | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

ZIP2 `<Pd>`., `<Pn>`., `<T>`.

if ![HaveSVE](https://www.example.com)() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;

**Low halves**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |   | Pm | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

ZIP1 `<Pd>`., `<Pn>`., `<T>`.

if ![HaveSVE](https://www.example.com)() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;

**Assembler Symbols**

`<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.

`<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

`<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

`<Pm>` Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

integer base = part * pairs;
for p = 0 to pairs-1
  Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, base+p, esize DIV 8];
  Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, base+p, esize DIV 8];
P[d] = result;
ZIP1, ZIP2 (vectors)

Interleave elements from two half vectors

Interleave alternating elements from the lowest or highest halves of the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: High halves, High halves (quadwords), Low halves and Low halves (quadwords)

High halves

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 1 size 1 Zm | 0 1 1 0 0 1 Zn | Zd |

ZIP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

High halves (quadwords)

(_FEAT_F64MM)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 1 1 0 1 Zm | 0 0 0 0 0 1 Zn | Zd |

ZIP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Low halves

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 1 size 1 Zm | 0 1 1 0 0 0 Zn | Zd |

ZIP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
**Low halves (quadwords)**

(FEAT_F64MM)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |

**ZIP1 <Zd>.Q, <Zn>.Q, <Zm>.Q**

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
if VL < esize * 2 then UNDEFINED;
integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Zeros();

integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];
Z[d] = result;
### Top-level encodings for A64

#### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td><strong>Reserved</strong></td>
</tr>
<tr>
<td>0001</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>0010</td>
<td><strong>SVE encodings</strong></td>
</tr>
<tr>
<td>0011</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>100x</td>
<td><strong>Data Processing -- Immediate</strong></td>
</tr>
<tr>
<td>101x</td>
<td><strong>Branches, Exception Generating and System instructions</strong></td>
</tr>
<tr>
<td>x1x0</td>
<td><strong>Loads and Stores</strong></td>
</tr>
<tr>
<td>x101</td>
<td><strong>Data Processing -- Register</strong></td>
</tr>
<tr>
<td>x111</td>
<td><strong>Data Processing -- Scalar Floating-Point and Advanced SIMD</strong></td>
</tr>
</tbody>
</table>

#### Reserved

These instructions are under the **top-level**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td><strong>UDF</strong></td>
</tr>
<tr>
<td></td>
<td>!= 000000000</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>!= 000000000</td>
<td></td>
<td><strong>UNALLOCATED</strong></td>
</tr>
</tbody>
</table>

#### SVE encodings

These instructions are under the **top-level**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>x1xxxx</td>
<td><strong>SVE Integer Multiply-Add - Predicated</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>000xxx</td>
<td><strong>SVE Integer Binary Arithmetic - Predicated</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>001xxx</td>
<td><strong>SVE Integer Reduction</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>100xxx</td>
<td><strong>SVE Bitwise Shift - Predicated</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>101xxx</td>
<td><strong>SVE Integer Unary Arithmetic - Predicated</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>000xxx</td>
<td><strong>SVE integer add/subtract vectors (unpredicated)</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>001xxx</td>
<td><strong>SVE Bitwise Logical - Unpredicated</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>0100xx</td>
<td><strong>SVE Index Generation</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>0101xx</td>
<td><strong>SVE Stack Allocation</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>011xxx</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>100xxx</td>
<td><strong>SVE Bitwise Shift - Unpredicated</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>1010xx</td>
<td><strong>SVE address generation</strong></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>1011xx</td>
<td><strong>SVE Integer Misc - Unpredicated</strong></td>
</tr>
<tr>
<td>Code</td>
<td>Type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>0x 1xxxx 11xxxx</td>
<td>SVE Element Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 00xxx</td>
<td>SVE Bitwise Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 01xxx</td>
<td>SVE Integer Wide Immediate - Predicated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 001000</td>
<td>DUP (indexed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 001001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 00101x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 0011x1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 001100</td>
<td>TBL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 001110</td>
<td>SVE Permute Vector - Unpredicated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 010xxx</td>
<td>SVE Permute Predicate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 011xxx</td>
<td>SVE permut vector elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 10xxx</td>
<td>SVE Permute Vector - Predicated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 1xxxx 11xxx</td>
<td>SEL (vectors)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>11 1xxxx 000xxx</td>
<td>SVE Permute Vector - Extract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0x 00xxx</td>
<td>SVE Integer Compare - Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0x 1xxxx</td>
<td>SVE integer compare with unsigned immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 0xxxx x0xxxx</td>
<td>SVE integer compare with signed immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 0xxxx 01xxx</td>
<td>SVE predicate logical operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 0xxxx 11xxx</td>
<td>SVE Propagate Break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 01xxx 01xxx</td>
<td>SVE Partition Break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 01xxx 11xxx</td>
<td>SVE Predicate Misc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 1xxxx 00xxx</td>
<td>SVE Integer Compare - Scalars</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 1xxxx 01xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 1xxxx 11xxx</td>
<td>SVE Integer Wide Immediate - Unpredicated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 100xx 10xxx</td>
<td>SVE Predicate Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 101xx 1000xx</td>
<td>SVE Inc/Dec by Predicate Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 101xx 1001xx</td>
<td>SVE Write FFR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 101xx 101xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x 11xxx 10xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x 0xxxxx 0xxxxx</td>
<td>SVE Integer Multiply-Add - Unpredicated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x 0xxxxx 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x 1xxxxx</td>
<td>SVE Multiply - Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x 0xxxxx 0xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x 0xxxxx 10xxx</td>
<td>SVE Misc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x 0xxxxx 11xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 0xxxxx 0xxxxx</td>
<td>FCMLA (vectors)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 00x1x 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 00000 100xxx</td>
<td>FCADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 00000 101xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 00000 11xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 00001 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 0010x 100xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 0010x 101xxx</td>
<td>SVE floating-point convert precision odd elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 0010x 11xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 01xxx 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 1xxxx x0x01x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x 1xxxx 00000x</td>
<td>SVE floating-point multiply-add (indexed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Instruction details</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 0001xx</td>
<td>SVE floating-point complex multiply-add (indexed)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 001000</td>
<td>SVE floating-point multiply (indexed)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 001001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 0011xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 01x0xx</td>
<td>SVE Floating Point Widening Multiply-Add - Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 01x1xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 10x00x</td>
<td>SVE Floating Point Widening Multiply-Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 10x1xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 110xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 111000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 111001</td>
<td>SVE floating point matrix multiply accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 11101x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 1xxxx 1111xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 0xxxx x1xxxx</td>
<td>SVE floating-point compare vectors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 0xxxx 000xxx</td>
<td>SVE floating-point arithmetic (unpredicated)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 0xxxx 100xxx</td>
<td>SVE Floating Point Arithmetic - Predicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 0xxxx 101xxx</td>
<td>SVE Floating Point Unary Operations - Predicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 000xx 001xxx</td>
<td>SVE floating-point recursive reduction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 001xx 0010xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 001xx 0011xx</td>
<td>SVE Floating Point Unary Operations - Unpredicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 010xx 001xxx</td>
<td>SVE Floating Point Compare - with Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 011xx 001xxx</td>
<td>SVE Floating Point Accumulating Reduction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 1x 1xxxx</td>
<td>SVE Floating Point Multiply-Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>SVE Memory - 32-bit Gather and Unsize Contiguous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>SVE Memory - Contiguous Load</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>SVE Memory - 64-bit Gather</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 0x0xxx</td>
<td>SVE Memory - Contiguous Store and Unsize Contiguous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 0x1xxx</td>
<td>SVE Memory - Non-temporal and Multi-register Store</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 1x0xxx</td>
<td>SVE Memory - Scatter with Optional Sign Extend</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 101xxx</td>
<td>SVE Memory - Scatter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 111xxx</td>
<td>SVE Memory - Contiguous Store with Immediate Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SVE Integer Multiply-Add - Predicated

These instructions are under SVE encodings.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
00000100 | 0 | op0 1
```

#### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE integer multiply-accumulate writing addend (predicated)</td>
</tr>
<tr>
<td>1</td>
<td>SVE integer multiply-add writing multiplicand (predicated)</td>
</tr>
</tbody>
</table>

### SVE integer multiply-accumulate writing addend (predicated)

These instructions are under SVE Integer Multiply-Add - Predicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 size | 0 | Zm | 0 1 | op | Pg | Zn | Zda
```
### SVE Integer Multiply-Add Writing Multiplicand (Predicated)

These instructions are under [SVE Integer Multiply-Add - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MLA</td>
</tr>
<tr>
<td>1</td>
<td>MLS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>1</th>
<th>1</th>
<th>op</th>
<th>Pg</th>
<th>Za</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SVE Integer Binary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MAD</td>
</tr>
<tr>
<td>1</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 1 0 0</th>
<th>0</th>
<th>op0</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000100</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>SVE integer add/subtract vectors (predicated)</td>
</tr>
<tr>
<td>01x</td>
<td>SVE integer min/max/difference (predicated)</td>
</tr>
<tr>
<td>100</td>
<td>SVE integer multiply vectors (predicated)</td>
</tr>
<tr>
<td>101</td>
<td>SVE integer divide vectors (predicated)</td>
</tr>
<tr>
<td>11x</td>
<td>SVE bitwise logical operations (predicated)</td>
</tr>
</tbody>
</table>

### SVE Integer Add/Subtract Vectors (Predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (vectors, predicated)</td>
</tr>
<tr>
<td>010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>SUBR (vectors)</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 1 0 0</th>
<th>opc</th>
<th>0 0 0</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 0 1</td>
<td>u</td>
<td>0 0 0</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>
### SVE integer multiply vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>U</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE integer divide vectors (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE bitwise logical operations (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE bitwise logical operations (predicated)

These instructions are under [SVE Integer Binary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ORR (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>EOR (vectors, predicated)</td>
</tr>
<tr>
<td>010</td>
<td>AND (vectors, predicated)</td>
</tr>
<tr>
<td>011</td>
<td>BIC (vectors, predicated)</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE Integer Reduction

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>SVE integer add reduction (predicated)</td>
</tr>
<tr>
<td>010</td>
<td>SVE integer min/max reduction (predicated)</td>
</tr>
<tr>
<td>0x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x</td>
<td>SVE constructive prefix (predicated)</td>
</tr>
<tr>
<td>110</td>
<td>SVE bitwise logical reduction (predicated)</td>
</tr>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE integer add reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>size</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

SVE integer min/max reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>size</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SVE constructive prefix (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>size</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE bitwise logical reduction (predicated)

These instructions are under SVE Integer Reduction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 | size 0 1 | opc 0 0 | Pg  | Zn | Vd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ORV</td>
</tr>
<tr>
<td>01</td>
<td>EORV</td>
</tr>
<tr>
<td>10</td>
<td>ANDV</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Bitwise Shift - Predicated

These instructions are under SVE encodings.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
00000100 | 0 | op0 | 100
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SVE bitwise shift by immediate (predicated)</td>
</tr>
<tr>
<td>10</td>
<td>SVE bitwise shift by vector (predicated)</td>
</tr>
<tr>
<td>11</td>
<td>SVE bitwise shift by wide elements (predicated)</td>
</tr>
</tbody>
</table>

SVE bitwise shift by immediate (predicated)

These instructions are under SVE Bitwise Shift - Predicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | tszh 0 0 | opc L U 1 0 0 | Pg tszl imm3 | Zdn
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (immediate, predicated)</td>
</tr>
<tr>
<td>00</td>
<td>LSR (immediate, predicated)</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>LSL (immediate, predicated)</td>
</tr>
<tr>
<td>01</td>
<td>ASRD</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE bitwise shift by vector (predicated)

These instructions are under SVE Bitwise Shift - Predicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | size 0 1 | R L U 1 0 0 | Pg Zm Zdn
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>ASR (vectors)</td>
</tr>
</tbody>
</table>
### SVE bitwise shift by wide elements (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LSR (vectors)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>LSL (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ASRR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LSRR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LSLR</td>
</tr>
</tbody>
</table>

### SVE Integer Unary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ASR (wide elements, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LSR (wide elements, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>LSL (wide elements, predicated)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE integer unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td></td>
<td>SVE integer unary operations (predicated)</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td></td>
<td>SVE bitwise unary operations (predicated)</td>
</tr>
</tbody>
</table>

### SVE Integer Unary Arithmetic - Predicated

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>R</th>
<th>L</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SXTB, SXTH, SXTW — SXTB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UXTB, UXTH, UXTW — UXTB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SXTB, SXTH, SXTW — SXTH</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UXTB, UXTH, UXTW — UXTH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SXTB, SXTH, SXTW — SXTW</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UXTB, UXTH, UXTW — UXTW</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>NEG</td>
</tr>
</tbody>
</table>
SVE bitwise unary operations (predicated)

These instructions are under SVE Integer Unary Arithmetic - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CLS</td>
</tr>
<tr>
<td>001</td>
<td>CLZ</td>
</tr>
<tr>
<td>010</td>
<td>CNT</td>
</tr>
<tr>
<td>011</td>
<td>CNOT</td>
</tr>
<tr>
<td>100</td>
<td>FABS</td>
</tr>
<tr>
<td>101</td>
<td>FNEG</td>
</tr>
<tr>
<td>110</td>
<td>NOT (vector)</td>
</tr>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE integer add/subtract vectors (unpredicated)

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (vectors, unpredicated)</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>SQADD (vectors)</td>
</tr>
<tr>
<td>101</td>
<td>UQADD (vectors)</td>
</tr>
<tr>
<td>110</td>
<td>SQSUB (vectors)</td>
</tr>
<tr>
<td>111</td>
<td>UQSUB (vectors)</td>
</tr>
</tbody>
</table>

SVE Bitwise Logical - Unpredicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 != 0</td>
<td>SVE bitwise logical operations (unpredicated)</td>
</tr>
</tbody>
</table>

SVE bitwise logical operations (unpredicated)

These instructions are under SVE Bitwise Logical - Unpredicated.
### Decode fields and Instruction Details

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>AND (vectors, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>ORR (vectors, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>EOR (vectors, unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>BIC (vectors, unpredicated)</td>
</tr>
</tbody>
</table>

### SVE Index Generation

These instructions are under SVE encodings.

![Index Generation Decode Fields](image)

### SVE Stack Allocation

These instructions are under SVE encodings.

![Stack Allocation Decode Fields](image)

### SVE stack frame adjustment

These instructions are under SVE Stack Allocation.

![Stack Frame Adjustment Decode Fields](image)

### SVE stack frame size

These instructions are under SVE Stack Allocation.

![Stack Frame Size Decode Fields](image)
SVE Bitwise Shift - Unpredicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE bitwise shift by wide elements (unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>SVE bitwise shift by immediate (unpredicated)</td>
</tr>
</tbody>
</table>

SVE bitwise shift by wide elements (unpredicated)

These instructions are under SVE Bitwise Shift - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LSL (wide elements, unpredicated)</td>
</tr>
</tbody>
</table>

SVE bitwise shift by immediate (unpredicated)

These instructions are under SVE Bitwise Shift - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (immediate, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (immediate, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LSL (immediate, unpredicated)</td>
</tr>
</tbody>
</table>

SVE address generation

These instructions are under SVE encodings.
### Decode fields  
<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ADR — Unpacked 32-bit signed offsets</td>
</tr>
<tr>
<td>01</td>
<td>ADR — Unpacked 32-bit unsigned offsets</td>
</tr>
<tr>
<td>1x</td>
<td>ADR — Packed offsets</td>
</tr>
</tbody>
</table>

### SVE Integer Misc - Unpredicated

These instructions are under [SVE encodings](#).

### SVE floating-point trig select coefficient

These instructions are under **SVE Integer Misc - Unpredicated**.

### SVE floating-point exponential accelerator

These instructions are under **SVE Integer Misc - Unpredicated**.

### SVE constructive prefix (unpredicated)

These instructions are under **SVE Integer Misc - Unpredicated**.
SVE Element Count

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>MOVPRFX (unpredicated)</td>
</tr>
<tr>
<td>00</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>01xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE saturating inc/dec vector by element count

These instructions are under SVE Element Count.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00x</td>
<td>SVE saturating inc/dec vector by element count</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>SVE element count</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>SVE inc/dec vector by element count</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>SVE inc/dec register by element count</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>11x</td>
<td>SVE saturating inc/dec register by element count</td>
</tr>
</tbody>
</table>

SVE saturating inc/dec vector by element count

These instructions are under SVE Element Count.

<table>
<thead>
<tr>
<th>size</th>
<th>D</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>SQINCH (vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>UQINCH (vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>SQDECH (vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>UQDECH (vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>SQINCW (vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>UQINCW (vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>SQDECW (vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>UQDECW (vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>SQINCD (vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>UQINCD (vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>SQDECD (vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>UQDECD (vector)</td>
</tr>
</tbody>
</table>
## SVE element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>op</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

## SVE inc/dec vector by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>D</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

## SVE inc/dec register by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>D</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

## SVE saturating inc/dec register by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>D</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
### SVE Bitwise Immediate

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size sf D U</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0</td>
<td>SQINCB — 32-bit</td>
</tr>
<tr>
<td>00 0 0 1</td>
<td>UQINCB — 32-bit</td>
</tr>
<tr>
<td>00 0 1 0</td>
<td>SQDECB — 32-bit</td>
</tr>
<tr>
<td>00 1 1 1</td>
<td>UDECB — 32-bit</td>
</tr>
<tr>
<td>00 1 0 0</td>
<td>SQINCB — 64-bit</td>
</tr>
<tr>
<td>00 1 0 1</td>
<td>UQINCB — 64-bit</td>
</tr>
<tr>
<td>00 1 1 0</td>
<td>SQDECB — 64-bit</td>
</tr>
<tr>
<td>00 1 1 1</td>
<td>UQDECB — 64-bit</td>
</tr>
<tr>
<td>01 0 0 0</td>
<td>SQINCH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 0 1</td>
<td>UQINCH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 1 0</td>
<td>SQDECH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 1 1</td>
<td>UQDECH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 1 0 0</td>
<td>SQINCH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 0 1</td>
<td>UQINCH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 1 0</td>
<td>SQDECH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 1 1</td>
<td>UQDECH (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 0 0 0</td>
<td>SQINCW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 0 1</td>
<td>UQINCW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 1 0</td>
<td>SQDECW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 1 1</td>
<td>UQDECW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 1 0 0</td>
<td>SQINCW (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 1 0 1</td>
<td>UQINCW (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 1 1 0</td>
<td>SQDECW (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 1 1 1</td>
<td>UQDECW (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 0 0 0</td>
<td>SQINCD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 0 1</td>
<td>UQINCD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 1 0</td>
<td>SQDECD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 1 1</td>
<td>UQDECD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 1 0 0</td>
<td>SQINCD (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 1 0 1</td>
<td>UQINCD (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 1 1 0</td>
<td>SQDECD (scalar) — 64-bit</td>
</tr>
<tr>
<td>11 1 1 1</td>
<td>UQDECD (scalar) — 64-bit</td>
</tr>
</tbody>
</table>

#### DUPM

These instructions are under SVE Bitwise Immediate.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>op1</td>
<td></td>
</tr>
<tr>
<td>11 00</td>
<td>DUPM</td>
</tr>
<tr>
<td>!= 11 00</td>
<td>SVE bitwise logical with immediate (unpredicated)</td>
</tr>
<tr>
<td>!= 00</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### SVE bitwise logical with immediate (unpredicated)

These instructions are under SVE Bitwise Immediate.
The following constraints also apply to this encoding: opc != 11 && opc != 11

### Decode fields OPC

<table>
<thead>
<tr>
<th>OPC</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ORR (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>EOR (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>AND (immediate)</td>
</tr>
</tbody>
</table>

### SVE Integer Wide Immediate - Predicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>OP0</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>SVE copy integer immediate (predicated)</td>
</tr>
<tr>
<td>10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FCPY</td>
</tr>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE copy integer immediate (predicated)

These instructions are under [SVE Integer Wide Immediate - Predicated](#).

<table>
<thead>
<tr>
<th>M</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPY (immediate, zeroing)</td>
</tr>
<tr>
<td>1</td>
<td>CPY (immediate, merging)</td>
</tr>
</tbody>
</table>

### SVE Permute Vector - Unpredicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>OP0</th>
<th>OP1</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>DUP (scalar)</td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>INSR (scalar)</td>
</tr>
<tr>
<td>00</td>
<td>x10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>xx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0xx</td>
<td>SVE unpack vector elements</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>INSR (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
## SVE unpack vector elements

These instructions are under **SVE Permute Vector - Unpredicated**.

<table>
<thead>
<tr>
<th>10</th>
<th>1x1</th>
<th>UNALLOCATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>000</td>
<td>REV (vector)</td>
</tr>
<tr>
<td>11</td>
<td>!= 000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>H</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>SUNPKHI, SUNPKLO — SUNPKLO</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>SUNPKHI, SUNPKLO — SUNPKHI</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>UUNPKHI, UUNPKLO — UUNPKLO</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>UUNPKHI, UUNPKLO — UUNPKHI</td>
</tr>
</tbody>
</table>

## SVE Permute Predicate

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>00000101</th>
<th>op0</th>
<th>1</th>
<th>op1</th>
<th>010</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>SVE unpack predicate elements</td>
</tr>
<tr>
<td>01</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0xxx</td>
<td>xxx0</td>
<td>0</td>
<td>SVE permute predicate elements</td>
<td></td>
</tr>
<tr>
<td>0xxx</td>
<td>xxx1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>0000</td>
<td>0</td>
<td>REV (predicate)</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>1000</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>x100</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>xx10</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>xxx1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x1x</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11xxx</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SVE unpack predicate elements

These instructions are under **SVE Permute Predicate**.

<table>
<thead>
<tr>
<th>00000101</th>
<th>H</th>
<th>Pn</th>
<th>Pd</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>H</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PUNPKHI, PUNPKLO — PUNPKLO</td>
</tr>
</tbody>
</table>
### SVE permute predicate elements

These instructions are under SVE Permute Predicate.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1</td>
<td>PUNPKHI, PUNPKLO — PUNPKHI</td>
</tr>
</tbody>
</table>

### SVE permute vector elements

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc 000</td>
<td>ZIP1, ZIP2 (predicates) — ZIP1</td>
</tr>
<tr>
<td>opc 001</td>
<td>ZIP1, ZIP2 (predicates) — ZIP2</td>
</tr>
<tr>
<td>opc 010</td>
<td>UZP1, UZP2 (predicates) — UZP1</td>
</tr>
<tr>
<td>opc 011</td>
<td>UZP1, UZP2 (predicates) — UZP2</td>
</tr>
<tr>
<td>opc 100</td>
<td>TRN1, TRN2 (predicates) — TRN1</td>
</tr>
<tr>
<td>opc 101</td>
<td>TRN1, TRN2 (predicates) — TRN2</td>
</tr>
<tr>
<td>opc 11x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Permute Vector - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 00000101</td>
<td>1 op0 op1 op2 10 op3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0 00000101</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>CPY (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>0 001</td>
<td>SVE extract element to general register</td>
</tr>
<tr>
<td>0 01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 100</td>
<td>CPY (scalar)</td>
</tr>
</tbody>
</table>
SVE extract element to general register

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (scalar)</td>
</tr>
</tbody>
</table>

SVE extract element to SIMD&FP scalar register

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

SVE reverse within elements

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>REVB, REVH, REVW — REVB</td>
</tr>
<tr>
<td>01</td>
<td>REVB, REVH, REVW — REVH</td>
</tr>
<tr>
<td>10</td>
<td>REVB, REVH, REVW — REVW</td>
</tr>
<tr>
<td>11</td>
<td>RBIT</td>
</tr>
</tbody>
</table>
SVE conditionally broadcast element to vector

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (vectors)</td>
</tr>
</tbody>
</table>

SVE conditionally extract element to SIMD&FP scalar

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

SVE conditionally extract element to general register

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (scalar)</td>
</tr>
</tbody>
</table>

SVE Permute Vector - Extract

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FXT</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Permute Vector - Segments

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FXT</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE permute vector segments

These instructions are under SVE Permute Vector - Segments.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE permute vector segments</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

SVE Integer Compare - Vectors

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZIP1, ZIP2 (vectors) — ZIP1</td>
<td>FEAT F64MM</td>
</tr>
<tr>
<td>ZIP1, ZIP2 (vectors) — ZIP2</td>
<td>FEAT F64MM</td>
</tr>
<tr>
<td>UZP1, UZP2 (vectors) — UZP1</td>
<td>FEAT F64MM</td>
</tr>
<tr>
<td>UZP1, UZP2 (vectors) — UZP2</td>
<td>FEAT F64MM</td>
</tr>
<tr>
<td>TRN1, TRN2 (vectors) — TRN1</td>
<td>FEAT F64MM</td>
</tr>
<tr>
<td>TRN1, TRN2 (vectors) — TRN2</td>
<td>FEAT F64MM</td>
</tr>
</tbody>
</table>

SVE integer compare vectors

These instructions are under SVE Integer Compare - Vectors.

<table>
<thead>
<tr>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE integer compare vectors</td>
<td></td>
</tr>
<tr>
<td>SVE integer compare with wide elements</td>
<td></td>
</tr>
</tbody>
</table>

SVE integer compare with wide elements

These instructions are under SVE Integer Compare - Vectors.
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>lt</th>
<th>ne</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPGE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPGT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPEQ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPO</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPR</td>
</tr>
</tbody>
</table>

#### SVE integer compare with unsigned immediate

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>size</th>
<th>imm7</th>
<th>lt</th>
<th>Pg</th>
<th>Zn</th>
<th>ne</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>lt</th>
<th>ne</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPEQ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPGT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPR</td>
</tr>
</tbody>
</table>

#### SVE integer compare with signed immediate

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>size</th>
<th>imm5</th>
<th>op</th>
<th>o2</th>
<th>Pg</th>
<th>Zn</th>
<th>ne</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>o2</th>
<th>ne</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPGE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPGT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPO</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### SVE predicate logical operations

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Pg</th>
<th>o2</th>
<th>Pn</th>
<th>o3</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AND (predicates)</td>
</tr>
</tbody>
</table>
## SVE Propagate Break

These instructions are under SVE encodings.

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>BIC (predicates)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>EOR (predicates)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>SEL (predicates)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ANDS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>BICS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>EORS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ORR (predicates)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>ORN (predicates)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>NOR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>NAND</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ORRS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>ORNS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NORS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>NANDS</td>
</tr>
</tbody>
</table>

### SVE propagate break from previous partition

These instructions are under SVE Propagate Break.

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE propagate break from previous partition</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

## SVE Partition Break

These instructions are under SVE encodings.

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BRKPA</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>BRKPB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>BRKPAS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>BRKPBS</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
**SVE propagate break to next partition**

These instructions are under **SVE Partition Break**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>SVE propagate break to next partition</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x000</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x1xx</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xx1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xxx1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE partition break condition**

These instructions are under **SVE Partition Break**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>SVE predicate test</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td>SVE predicate first active</td>
</tr>
</tbody>
</table>

**SVE Predicate Misc**

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>SVE predicate test</td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0xx1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0xxx</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>0</td>
<td></td>
<td>0</td>
<td>SVE predicate first active</td>
</tr>
</tbody>
</table>
### SVE predicate test

These instructions are under **SVE Predicate Misc**.

| 1000 | 000 | != 00 | 0 | UNALLOCATED |
| 1000 | 100 | 10 | 0000 | 0 | SVE predicate zero |
| 1000 | 100 | 10 | != 0000 | 0 | UNALLOCATED |
| 1000 | 110 | 00 | 0 | SVE predicate read from FFR (predicated) |
| 1001 | 000 | 0x | 0 | UNALLOCATED |
| 1001 | 000 | 10 | 0 | PNEXT |
| 1001 | 000 | 11 | 0 | UNALLOCATED |
| 1001 | 100 | 10 | 0 | UNALLOCATED |
| 1001 | 110 | 00 | 0000 | 0 | SVE predicate read from FFR (unpredicated) |
| 1001 | 110 | 00 | != 0000 | 0 | UNALLOCATED |
| 100x | 010 | 0 | UNALLOCATED |
| 100x | 100 | 0x | 0 | SVE predicate initialize |
| 100x | 100 | 11 | 0 | UNALLOCATED |
| 100x | 110 | 0! | 00 | 0 | UNALLOCATED |
| 100x | xx1 | 0 | UNALLOCATED |
| 110x | 0 | UNALLOCATED |
| 1x1x | 0 | UNALLOCATED |

| 1 | UNALLOCATED |

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0000</td>
<td>PTEST</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate first active

These instructions are under **SVE Predicate Misc**.

| 1000 | 000 | != 00 | 0 | UNALLOCATED |
| 1000 | 100 | 10 | 0000 | 0 | SVE predicate zero |
| 1000 | 100 | 10 | != 0000 | 0 | UNALLOCATED |
| 1000 | 110 | 00 | 0 | SVE predicate read from FFR (predicated) |
| 1001 | 000 | 0x | 0 | UNALLOCATED |
| 1001 | 000 | 10 | 0 | PNEXT |
| 1001 | 000 | 11 | 0 | UNALLOCATED |
| 1001 | 100 | 10 | 0 | UNALLOCATED |
| 1001 | 110 | 00 | 0000 | 0 | SVE predicate read from FFR (unpredicated) |
| 1001 | 110 | 00 | != 0000 | 0 | UNALLOCATED |
| 100x | 010 | 0 | UNALLOCATED |
| 100x | 100 | 0x | 0 | SVE predicate initialize |
| 100x | 100 | 11 | 0 | UNALLOCATED |
| 100x | 110 | 0! | 00 | 0 | UNALLOCATED |
| 100x | xx1 | 0 | UNALLOCATED |
| 110x | 0 | UNALLOCATED |
| 1x1x | 0 | UNALLOCATED |

| 1 | UNALLOCATED |

#### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>PFIRST</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate zero

These instructions are under **SVE Predicate Misc**.
### SVE predicate read from FFR (predicated)

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>PFALSE</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>RDFFR (predicated)</td>
</tr>
<tr>
<td>0 1</td>
<td>RDFFRS</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate read from FFR (unpredicated)

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>RDFFR (unpredicated)</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate initialize

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PTRUE</td>
</tr>
<tr>
<td>1</td>
<td>PTRUES</td>
</tr>
</tbody>
</table>

### SVE Integer Compare - Scalars

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PTRUE</td>
</tr>
<tr>
<td>1</td>
<td>PTRUES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>SVE integer compare scalar count and limit</td>
</tr>
</tbody>
</table>
SVE conditionally terminate scalars

These instructions are under SVE Integer Compare - Scalars.

| 1 | 000 | 0000 | SVE conditionally terminate scalars |
| 1 | 000 | != 0000 | UNALLOCATED |
| 1 | != 000 | UNALLOCATED |

SVE integer compare scalar count and limit

These instructions are under SVE Integer Compare - Scalars.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-------------------|
| 0 0 1 0 0 1 0 1 | size | 1 | Rm | 0 0 0 | sf | U | lt | Rn | eq | Pd |

**Decode fields**

<table>
<thead>
<tr>
<th>U</th>
<th>lt</th>
<th>eq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>WHILELT</td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>WHILELE</td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>WHILELO</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>WHILELS</td>
<td></td>
</tr>
</tbody>
</table>

SVE conditionally terminate scalars

These instructions are under SVE Integer Compare - Scalars.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-------------------|
| 0 0 1 0 0 1 0 1 | op | sz | 1 | Rm | 0 0 1 0 0 0 | Rn | ne | 0 0 0 0 |

**Decode fields**

<table>
<thead>
<tr>
<th>op</th>
<th>ne</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0</td>
<td>CTERMEQ, CTERMNE — CTERMEQ</td>
</tr>
<tr>
<td>1 1</td>
<td>CTERMEQ, CTERMNE — CTERMNE</td>
</tr>
</tbody>
</table>

SVE Integer Wide Immediate - Unpredicated

These instructions are under SVE encodings.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-------------------|
| 00100101 | 1 | op0 | p|pl | 11 |

**Decode fields**

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE integer add/subtract immediate (unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>SVE integer min/max immediate (unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>SVE integer multiply immediate (unpredicated)</td>
</tr>
<tr>
<td>11 0</td>
<td>SVE broadcast integer immediate (unpredicated)</td>
</tr>
<tr>
<td>11 1</td>
<td>SVE broadcast floating-point immediate (unpredicated)</td>
</tr>
</tbody>
</table>

SVE integer add/subtract immediate (unpredicated)

These instructions are under SVE Integer Wide Immediate - Unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-------------------|
| 0 0 1 0 0 1 0 1 | size | 1 0 0 | opc | 1 1 | sh | imm8 | Zdn |
### SVE Integer Min/Max Immediate (Unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ADD (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>SUBR (immediate)</td>
</tr>
<tr>
<td>100</td>
<td>SQADD (immediate)</td>
</tr>
<tr>
<td>101</td>
<td>UQADD (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>SQSUB (immediate)</td>
</tr>
<tr>
<td>111</td>
<td>UQSUB (immediate)</td>
</tr>
</tbody>
</table>

#### Encoding

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | opc | 1  | 1  | o2 | imm8 | Zdn |

### SVE Integer Multiply Immediate (Unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>0xx</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Encoding

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | size | 1  | 1  | 0  | opc | 1  | 1  | o2 | imm8 | Zdn |

### SVE Broadcast Integer Immediate (Unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>MUL (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Encoding

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | size | 1  | 1  | 1  | opc | 0  | 1  | 1  | sh | imm8 | Zd |

### SVE Broadcast Integer Immediate (Unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>DUP (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE broadcast floating-point immediate (unpredicated)

These instructions are under SVE Integer Wide Immediate - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc o2</td>
<td></td>
</tr>
<tr>
<td>00 0</td>
<td>FDUP</td>
</tr>
<tr>
<td>00 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Predicate Count

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SVE predicate count</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE predicate count

These instructions are under SVE Predicate Count.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>CNTP</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Inc/Dec by Predicate Count

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>SVE saturating inc/dec vector by predicate count</td>
</tr>
<tr>
<td>0 1</td>
<td>SVE saturating inc/dec register by predicate count</td>
</tr>
<tr>
<td>1 0</td>
<td>SVE inc/dec vector by predicate count</td>
</tr>
<tr>
<td>1 1</td>
<td>SVE inc/dec register by predicate count</td>
</tr>
</tbody>
</table>
SVE saturating inc/dec vector by predicate count

These instructions are under SVE Inc/Dec by Predicate Count.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D U opc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1X</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>SQINCP (vector)</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>UQINCP (vector)</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>SQDECP (vector)</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>UQDECP (vector)</td>
</tr>
</tbody>
</table>

SVE saturating inc/dec register by predicate count

These instructions are under SVE Inc/Dec by Predicate Count.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D U sf op</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>SQINCP (scalar) — 32-bit</td>
</tr>
<tr>
<td>0 0 1 0 0</td>
<td>SQINCP (scalar) — 64-bit</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>UQINCP (scalar) — 32-bit</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>UQINCP (scalar) — 64-bit</td>
</tr>
<tr>
<td>1 0 0 0 0</td>
<td>SQDECP (scalar) — 32-bit</td>
</tr>
<tr>
<td>1 0 0 1 0</td>
<td>SQDECP (scalar) — 64-bit</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>UDECP (scalar) — 32-bit</td>
</tr>
<tr>
<td>1 1 0 1 0</td>
<td>UDECP (scalar) — 64-bit</td>
</tr>
</tbody>
</table>

SVE inc/dec vector by predicate count

These instructions are under SVE Inc/Dec by Predicate Count.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D opc</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>INCP (vector)</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>DECP (vector)</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE inc/dec register by predicate count

These instructions are under SVE Inc/Dec by Predicate Count.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D opc2 Pm</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 0 0 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE Write FFR

These instructions are under [SVE encodings](#).

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 00100101 | 101 | op0 | op1 | 1001 | op2 | op3 | op4 |
```

### SVE FFR write from predicate

These instructions are under [SVE Write FFR](#).

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 1 0 0 1 0 1 | opc | 1 0 1 0 0 0 1 0 0 1 0 0 0 | Pn | 0 0 0 0 0 |
```

### SVE FFR initialise

These instructions are under [SVE Write FFR](#).

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 1 0 0 1 0 1 | opc | 1 0 1 1 0 0 1 0 0 1 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
```
## SVE Integer Multiply-Add - Unpredicated

These instructions are under SVE encodings.

```
   01000100 | 0 | 0 0 p0 01 02
```

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td></td>
<td>SVE integer dot product (unpredicated)</td>
</tr>
<tr>
<td>0</td>
<td>!= 000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>0</td>
<td>SVE mixed sign dot product</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE integer dot product (unpredicated)

These instructions are under SVE Integer Multiply-Add - Unpredicated.

```
   01000100 | 0 | 0 0 0 0 0 0 0 | size | 0 | Zm | 0 0 0 0 0 | U | Zn | Zda
```

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SDOT (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>UDOT (vectors)</td>
</tr>
</tbody>
</table>

### SVE mixed sign dot product

These instructions are under SVE Integer Multiply-Add - Unpredicated.

```
   01000100 | 0 | 0 0 0 0 0 0 | size | 0 | Zm | 0 1 1 1 1 0 | Zn | Zda
```

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>USDOT (vectors)</td>
<td>FEAT_18MM</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

## SVE Multiply - Indexed

These instructions are under SVE encodings.

```
   01000100 | 1 | 0 0 p0 01
```

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>SVE integer dot product (indexed)</td>
</tr>
<tr>
<td>000</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000</td>
<td>11</td>
<td>SVE mixed sign dot product (indexed)</td>
</tr>
<tr>
<td>!= 000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE integer dot product (indexed)

These instructions are under SVE Multiply - Indexed.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SDOT (indexed)</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>UDOT (indexed)</td>
<td>-</td>
</tr>
</tbody>
</table>

SVE mixed sign dot product (indexed)

These instructions are under SVE Multiply - Indexed.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>USDOT (indexed)</td>
<td>FEAT I8MM</td>
</tr>
<tr>
<td>11</td>
<td>SUDOT</td>
<td>FEAT I8MM</td>
</tr>
</tbody>
</table>

SVE Misc

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>010x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>SVE integer matrix multiply accumulate</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

SVE integer matrix multiply accumulate

These instructions are under SVE Misc.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SMMLA</td>
<td>FEAT I8MM</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>USMMLA</td>
<td>FEAT I8MM</td>
</tr>
</tbody>
</table>
### SVE floating-point convert precision odd elements

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>UMMLA</td>
<td>FEAT_18MM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc opc2</td>
<td>BFCVTNT</td>
<td>FEAT_BF16</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply-add (indexed)

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>BFCVTNT</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE floating-point complex multiply-add (indexed)

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>FMLA (indexed) — half-precision</td>
</tr>
<tr>
<td>0x</td>
<td>FMLS (indexed) — half-precision</td>
</tr>
<tr>
<td>10</td>
<td>FMLA (indexed) — single-precision</td>
</tr>
<tr>
<td>10</td>
<td>FMLS (indexed) — single-precision</td>
</tr>
<tr>
<td>11</td>
<td>FMLA (indexed) — double-precision</td>
</tr>
<tr>
<td>11</td>
<td>FMLS (indexed) — double-precision</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply (indexed)

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>FCMLA (indexed) — half-precision</td>
</tr>
<tr>
<td>11</td>
<td>FCMLA (indexed) — single-precision</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply (indexed)

These instructions are under **SVE encodings**.
### SVE Floating Point Widening Multiply-Add - Indexed

These instructions are under **SVE encodings**.

### SVE BF16 floating-point dot product (indexed)

These instructions are under **SVE Floating Point Widening Multiply-Add - Indexed**.

### SVE floating-point multiply-add long (indexed)

These instructions are under **SVE Floating Point Widening Multiply-Add - Indexed**.

### SVE Floating Point Widening Multiply-Add

These instructions are under **SVE encodings**.
SVE BFloat16 floating-point dot product

These instructions are under SVE Floating Point Widening Multiply-Add.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1</td>
<td>BFDOT (vectors)</td>
<td>FEAT_BF16</td>
</tr>
</tbody>
</table>

SVE floating-point multiply-add long

These instructions are under SVE Floating Point Widening Multiply-Add.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 0 2 1</td>
<td>BFMLALB (vectors)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1 0 1 0 2 1</td>
<td>BFMLALT (vectors)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

SVE floating point matrix multiply accumulate

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1</td>
<td>BFMMMLA</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1 0</td>
<td>FMMLA — 32-bit element</td>
<td>FEAT_F32MM</td>
</tr>
<tr>
<td>1 1</td>
<td>FMMLA — 64-bit element</td>
<td>FEAT_F64MM</td>
</tr>
</tbody>
</table>

SVE floating-point compare vectors

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>FCM&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>0 1</td>
<td>BM</td>
</tr>
<tr>
<td>1 0 1</td>
<td>PG</td>
</tr>
<tr>
<td>1 1 0 0 1</td>
<td>Zm</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>opc</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>size</td>
</tr>
<tr>
<td>0 0 0 1 0 1</td>
<td>Zm</td>
</tr>
<tr>
<td>0 0 0 1 0 2</td>
<td>Pg</td>
</tr>
<tr>
<td>0 1 0 0 1</td>
<td>Zm</td>
</tr>
<tr>
<td>0 0 0 1 0 3</td>
<td>o3</td>
</tr>
<tr>
<td>0 0 0 1 0 4</td>
<td>Zn</td>
</tr>
<tr>
<td>0 0 0 1 0 5</td>
<td>o3</td>
</tr>
</tbody>
</table>

The top-level encodings for A64
### SVE floating-point arithmetic (unpredicated)

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>FCM&lt;cc&gt; (vectors) — FCMGT</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMEQ</td>
</tr>
<tr>
<td>0 1 1</td>
<td>FCM&lt;cc&gt; (vectors) — FCMNE</td>
</tr>
<tr>
<td>1 0 0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMUO</td>
</tr>
<tr>
<td>1 0 1</td>
<td>FAC&lt;cc&gt; — FACGE</td>
</tr>
<tr>
<td>1 1 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1 1</td>
<td>FAC&lt;cc&gt; — FACGT</td>
</tr>
</tbody>
</table>

### SVE Floating Point Arithmetic - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>001</td>
<td>FSUB (vectors, unpredicated)</td>
</tr>
<tr>
<td>010</td>
<td>FMUL (vectors, unpredicated)</td>
</tr>
<tr>
<td>011</td>
<td>FTSMUL</td>
</tr>
<tr>
<td>10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FRECPS</td>
</tr>
<tr>
<td>111</td>
<td>FRSQRTS</td>
</tr>
</tbody>
</table>

### SVE floating-point arithmetic (predicated)

These instructions are under SVE Floating Point Arithmetic - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SVE floating-point arithmetic (predicated)</td>
</tr>
<tr>
<td>10 000</td>
<td>FTMAD</td>
</tr>
<tr>
<td>10 != 000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 0000</td>
<td>SVE floating-point arithmetic with immediate (predicated)</td>
</tr>
<tr>
<td>11 != 0000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE floating-point arithmetic with immediate (predicated)

These instructions are under SVE Floating Point Arithmetic - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FADD (vectors, predicated)</td>
</tr>
<tr>
<td>0001</td>
<td>FSUB (vectors, predicated)</td>
</tr>
<tr>
<td>0010</td>
<td>FMUL (vectors, predicated)</td>
</tr>
</tbody>
</table>
SVE floating-point arithmetic with immediate (predicated)

These instructions are under **SVE Floating Point Arithmetic - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>FSUBR (vectors)</td>
</tr>
<tr>
<td>0100</td>
<td>FMAXNM (vectors)</td>
</tr>
<tr>
<td>0101</td>
<td>FMINNM (vectors)</td>
</tr>
<tr>
<td>0110</td>
<td>FMAX (vectors)</td>
</tr>
<tr>
<td>0111</td>
<td>FMIN (vectors)</td>
</tr>
<tr>
<td>1000</td>
<td>FABD</td>
</tr>
<tr>
<td>1001</td>
<td>FSACLE</td>
</tr>
<tr>
<td>1010</td>
<td>FMULX</td>
</tr>
<tr>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100</td>
<td>FDIVR</td>
</tr>
<tr>
<td>1101</td>
<td>FDIV</td>
</tr>
<tr>
<td>111x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Floating Point Unary Operations - Predicated

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADD (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>FSUB (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>FMUL (immediate)</td>
</tr>
<tr>
<td>011</td>
<td>FSUBR (immediate)</td>
</tr>
<tr>
<td>100</td>
<td>FMAXNM (immediate)</td>
</tr>
<tr>
<td>101</td>
<td>FMINNM (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>FMAX (immediate)</td>
</tr>
<tr>
<td>111</td>
<td>FMIN (immediate)</td>
</tr>
</tbody>
</table>

SVE floating-point round to integral value

These instructions are under **SVE Floating Point Unary Operations - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>SVE floating-point round to integral value</td>
</tr>
<tr>
<td>010</td>
<td>SVE floating-point convert precision</td>
</tr>
<tr>
<td>011</td>
<td>SVE floating-point unary operations</td>
</tr>
<tr>
<td>10x</td>
<td>SVE integer convert to floating-point</td>
</tr>
<tr>
<td>11x</td>
<td>SVE floating-point convert to integer</td>
</tr>
</tbody>
</table>
### SVE floating-point convert precision

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10 00</td>
<td>FCVT — single-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>10 01</td>
<td>FCVT — half-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>10 10</td>
<td>BFCVT FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11 00</td>
<td>FCVT — double-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>11 01</td>
<td>FCVT — half-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>11 10</td>
<td>FCVT — double-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>11 11</td>
<td>FCVT — single-precision to double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

### SVE floating-point unary operations

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FRECPX</td>
</tr>
<tr>
<td>01</td>
<td>FSQRT</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE integer convert to floating-point

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).
### SVE floating-point convert to integer

These instructions are under [SVE Floating Point Unary Operations - Predicated](#).

![Top-level encodings for A64](#)

---

#### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>SCVTFS — 16-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>UCVTF — 16-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>SCVTFS — 32-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>UCVTF — 32-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>SCVTFS — 64-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>UCVTF — 64-bit to half-precision</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>SCVTFS — 32-bit to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>UCVTF — 32-bit to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>SCVTFS — 32-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>1</td>
<td>UCVTF — 32-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>SCVTFS — 64-bit to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>UCVTF — 64-bit to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>SCVTFS — 64-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>UCVTF — 64-bit to double-precision</td>
</tr>
</tbody>
</table>

#### Instruction Details

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>FCVTZS — half-precision to 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>FCVTZU — half-precision to 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>FCVTZS — half-precision to 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>FCVTZU — half-precision to 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>FCVTZS — half-precision to 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>FCVTZU — half-precision to 64-bit</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>FCVTZS — single-precision to 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>FCVTZU — single-precision to 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>FCVTZS — double-precision to 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>1</td>
<td>FCVTZU — double-precision to 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>FCVTZS — single-precision to 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>FCVTZU — single-precision to 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>FCVTZS — double-precision to 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>FCVTZU — double-precision to 64-bit</td>
</tr>
</tbody>
</table>
SVE floating-point recursive reduction

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADDV</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>FMAXNMV</td>
</tr>
<tr>
<td>101</td>
<td>FMINNMV</td>
</tr>
<tr>
<td>110</td>
<td>FMAXV</td>
</tr>
<tr>
<td>111</td>
<td>FMINV</td>
</tr>
</tbody>
</table>

SVE Floating Point Unary Operations - Unpredicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100101</td>
<td>001</td>
</tr>
<tr>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td>op0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE floating-point reciprocal estimate (unpredicated)</td>
</tr>
<tr>
<td>!= 00</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE floating-point reciprocal estimate (unpredicated)

These instructions are under SVE Floating Point Unary Operations - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FRECPE</td>
</tr>
<tr>
<td>111</td>
<td>FRSORTE</td>
</tr>
</tbody>
</table>

SVE Floating Point Compare - with Zero

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100101</td>
<td>010 op0 001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE floating-point compare with zero</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE floating-point compare with zero

These instructions are under [SVE Floating Point Compare - with Zero](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>FCM&lt;cc&gt; (zero) — FCMGE</td>
</tr>
<tr>
<td>0 0 1</td>
<td>FCM&lt;cc&gt; (zero) — FCMGT</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FCM&lt;cc&gt; (zero) — FCMLT</td>
</tr>
<tr>
<td>0 1 1</td>
<td>FCM&lt;cc&gt; (zero) — FCMLE</td>
</tr>
<tr>
<td>1 1 0</td>
<td>FCM&lt;cc&gt; (zero) — FCMNE</td>
</tr>
</tbody>
</table>

---

SVE Floating Point Accumulating Reduction

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 x x x</td>
<td>SVE floating-point serial reduction (predicated)</td>
</tr>
<tr>
<td>1 x x x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

SVE floating-point serial reduction (predicated)

These instructions are under [SVE Floating Point Accumulating Reduction](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FADDA</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

SVE Floating Point Multiply-Add

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE floating-point multiply-accumulate writing addend</td>
</tr>
<tr>
<td>1</td>
<td>SVE floating-point multiply-accumulate writing multiplicand</td>
</tr>
</tbody>
</table>
SVE floating-point multiply-accumulate writing addend

These instructions are under [SVE Floating Point Multiply-Add](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FMLA (vectors)</td>
</tr>
<tr>
<td>01</td>
<td>FMLS (vectors)</td>
</tr>
<tr>
<td>10</td>
<td>FNMLA</td>
</tr>
<tr>
<td>11</td>
<td>FNMLS</td>
</tr>
</tbody>
</table>

SVE floating-point multiply-accumulate writing multiplicand

These instructions are under [SVE Floating Point Multiply-Add](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FMAD</td>
</tr>
<tr>
<td>01</td>
<td>FMSB</td>
</tr>
<tr>
<td>10</td>
<td>FNMAF</td>
</tr>
<tr>
<td>11</td>
<td>FNMSB</td>
</tr>
</tbody>
</table>

SVE Memory - 32-bit Gather and Unsized Contiguous

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x1</td>
<td>0xx</td>
<td>0</td>
<td>SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)</td>
</tr>
<tr>
<td>00</td>
<td>x1</td>
<td>0xx</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>x1</td>
<td>0xx</td>
<td>1</td>
<td>SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>x1</td>
<td>0xx</td>
<td>1</td>
<td>SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)</td>
</tr>
<tr>
<td>11</td>
<td>0x</td>
<td>000</td>
<td>0</td>
<td>LDR (predicate)</td>
</tr>
<tr>
<td>11</td>
<td>0x</td>
<td>000</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>0x</td>
<td>010</td>
<td></td>
<td>LDR (vector)</td>
</tr>
<tr>
<td>11</td>
<td>0x</td>
<td>0x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1x</td>
<td>0xx</td>
<td>0</td>
<td>SVE contiguous prefetch (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>1x</td>
<td>0xx</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>x0</td>
<td>0xx</td>
<td></td>
<td>SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>00</td>
<td>10x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td>0</td>
<td></td>
<td>SVE contiguous prefetch (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>111</td>
<td>0</td>
<td></td>
<td>SVE 32-bit gather prefetch (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>11x</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>1xx</td>
<td></td>
<td></td>
<td>SVE 32-bit gather load (vector plus immediate)</td>
</tr>
<tr>
<td>1x</td>
<td>1xx</td>
<td></td>
<td></td>
<td>SVE load and broadcast element</td>
</tr>
</tbody>
</table>
### SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE contiguous prefetch (scalar plus immediate)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus immediate)</td>
</tr>
</tbody>
</table>
SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

The following constraints also apply to this encoding: opc != 11 && opc != 11

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc U ff</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00 0 0</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00 1 0</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00 1 1</td>
<td>LDFF1B (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE contiguous prefetch (scalar plus scalar)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc msz U</td>
<td>PRFB (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>PRFH (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>PRFW (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>PRFD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather prefetch (vector plus immediate)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc msz</td>
<td>PRFB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFD (vector plus immediate)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather load (vector plus immediate)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.
### SVE load and broadcast element

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

### SVE Memory - Contiguous Load

These instructions are under [SVE encodings](#).

---

<table>
<thead>
<tr>
<th>Decode fields msz U ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00 0 1</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00 1 0</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00 1 1</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01 0 0</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (vector plus immediate)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields dtypeh dtypel</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>LD1RB — 8-bit element</td>
</tr>
<tr>
<td>00 01</td>
<td>LD1RB — 16-bit element</td>
</tr>
<tr>
<td>00 10</td>
<td>LD1RB — 32-bit element</td>
</tr>
<tr>
<td>00 11</td>
<td>LD1RB — 64-bit element</td>
</tr>
<tr>
<td>01 00</td>
<td>LD1RSW</td>
</tr>
<tr>
<td>01 01</td>
<td>LD1RH — 16-bit element</td>
</tr>
<tr>
<td>01 10</td>
<td>LD1RH — 32-bit element</td>
</tr>
<tr>
<td>01 11</td>
<td>LD1RH — 64-bit element</td>
</tr>
<tr>
<td>10 00</td>
<td>LD1RSH — 64-bit element</td>
</tr>
<tr>
<td>10 01</td>
<td>LD1RSH — 32-bit element</td>
</tr>
<tr>
<td>10 10</td>
<td>LD1RW — 32-bit element</td>
</tr>
<tr>
<td>10 11</td>
<td>LD1RW — 64-bit element</td>
</tr>
<tr>
<td>11 00</td>
<td>LD1RSB — 64-bit element</td>
</tr>
<tr>
<td>11 01</td>
<td>LD1RSB — 32-bit element</td>
</tr>
<tr>
<td>11 10</td>
<td>LD1RSB — 16-bit element</td>
</tr>
<tr>
<td>11 11</td>
<td>LD1RD</td>
</tr>
</tbody>
</table>

### Instruction details

- **SVE contiguous non-temporal load (scalar plus immediate)**
SVE contiguous non-temporal load (scalar plus scalar)

These instructions are under **SVE Memory - Contiguous Load**.

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>LDNT1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDNT1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>LDNT1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>
```

SVE load multiple structures (scalar plus immediate)

These instructions are under **SVE Memory - Contiguous Load**.

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>LD2B (scalar plus immediate)</td>
</tr>
</tbody>
</table>
```

The following constraints also apply to this encoding: opc != 00 && opc != 00
### SVE load multiple structures (scalar plus scalar)

These instructions are under **SVE Memory - Contiguous Load**.

<table>
<thead>
<tr>
<th>msz</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>10</td>
<td>LD3B (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>LD4B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD2H (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>LD3H (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>LD4H (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>LD2W (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>LD3W (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>LD4W (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>LD2D (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>LD3D (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>LD4D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00

### SVE load and broadcast quadword (scalar plus immediate)

These instructions are under **SVE Memory - Contiguous Load**.

<table>
<thead>
<tr>
<th>msz</th>
<th>ssz</th>
<th>imm4</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LD1ROB (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LD1ROH (scalar plus immediate)</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LD1ROH (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
</tbody>
</table>
### SVE contiguous load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 00</td>
<td>LD1ROW (scalar plus immediate)</td>
<td></td>
</tr>
<tr>
<td>10 01</td>
<td>LD1ROW (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
<tr>
<td>11 00</td>
<td>LD1ROD (scalar plus immediate)</td>
<td></td>
</tr>
<tr>
<td>11 01</td>
<td>LD1ROD (scalar plus immediate)</td>
<td>FEAT_F64MM</td>
</tr>
</tbody>
</table>

### SVE contiguous non-fault load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LD1B (scalar plus immediate) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LD1B (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LD1B (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LD1B (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LD1SW (scalar plus immediate)</td>
</tr>
<tr>
<td>0101</td>
<td>LD1H (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LD1H (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LD1H (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LD1SH (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LD1SH (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LD1W (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LD1W (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LD1SB (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LD1SB (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LD1SB (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LD1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LDNF1B — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LDNF1B — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LDNF1B — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LDNF1B — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LDNF1SW</td>
</tr>
<tr>
<td>0101</td>
<td>LDNF1H — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LDNF1H — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LDNF1H — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LDNF1SH — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LDNF1SH — 32-bit element</td>
</tr>
</tbody>
</table>
**Decode fields**

<table>
<thead>
<tr>
<th>dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>LDNF1W — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LDNF1W — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LDNF1SB — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LDNF1SB — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LDNF1SB — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LDNF1D</td>
</tr>
</tbody>
</table>

**SVE load and broadcast quadword (scalar plus scalar)**

These instructions are under SVE Memory - Contiguous Load.

**Decode fields**

<table>
<thead>
<tr>
<th>msz</th>
<th>ssz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>LD1ROB (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1ROB (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1ROH (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1ROH (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>LD1ROW (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>LD1ROW (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>LD1ROD (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>LD1ROD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

**SVE contiguous load (scalar plus scalar)**

These instructions are under SVE Memory - Contiguous Load.

**Decode fields**

<table>
<thead>
<tr>
<th>dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LD1B (scalar plus scalar) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LD1B (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LD1B (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LD1B (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LD1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>0101</td>
<td>LD1H (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LD1H (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LD1H (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LD1SH (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LD1SH (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LD1W (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LD1W (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LD1SB (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LD1SB (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LD1SB (scalar plus scalar) — 16-bit element</td>
</tr>
</tbody>
</table>
SVE contiguous first-fault load (scalar plus scalar)

These instructions are under **SVE Memory - Contiguous Load**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>LD1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE Memory - 64-bit Gather

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LDFF1B (scalar plus scalar) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LDFF1B (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LDFF1B (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LDFF1B (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LDFF1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>0101</td>
<td>LDFF1H (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LDFF1H (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LDFF1H (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LDFF1SH (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LDFF1SH (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LDFF1W (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LDFF1W (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LDFF1SB (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LDFF1SB (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LDFF1SB (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LDFF1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE Memory - 64-bit Gather (continued)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 64-bit gather prefetch (vector plus immediate)</td>
</tr>
<tr>
<td>00, x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01, 10x</td>
<td>SVE 64-bit gather load (scalar plus 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>00, 110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00, 111</td>
<td>SVE 64-bit gather prefetch (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit gather load (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)</td>
</tr>
<tr>
<td>x0</td>
<td>SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)</td>
</tr>
</tbody>
</table>
SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit gather load (scalar plus 64-bit scaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00

SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>
The following constraints also apply to this encoding: opc != 00 && opc != 00

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LD1D (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>LDFF1D (vector plus immediate)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather prefetch (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>msz</th>
<th>Uff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>PRFB (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>PRFD (vector plus immediate)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>msz</th>
<th>Uff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>0</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>1</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>1</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>0</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>1</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>1</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>1</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>0</td>
<td>LD1SW (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1</td>
<td>LDFF1SW (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1</td>
<td>LD1W (vector plus immediate)</td>
</tr>
</tbody>
</table>
### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>LD1D (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>LDFF1D (vector plus immediate)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDFF1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDFF1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
</tbody>
</table>
### SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 1 1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE contiguous store (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Store and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op0 0 0</td>
<td>STR (predicate)</td>
</tr>
<tr>
<td>op0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op0 1</td>
<td>STR (vector)</td>
</tr>
<tr>
<td>op0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>!= 110 1</td>
<td>SVE contiguous store (scalar plus scalar)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 110 && opc != 110

### SVE Memory - Non-temporal and Multi-register Store

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>ST1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01x</td>
<td>ST1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10x</td>
<td>ST1W (scalar plus scalar)</td>
</tr>
<tr>
<td>111 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111 1</td>
<td>ST1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous non-temporal store (scalar plus scalar)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 1</td>
<td>SVE contiguous non-temporal store (scalar plus scalar)</td>
</tr>
<tr>
<td>!= 00 1</td>
<td>SVE store multiple structures (scalar plus scalar)</td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE store multiple structures (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

The following constraints also apply to this encoding: opc != 00 && opc != 00

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01</td>
<td>ST2B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 10</td>
<td>ST3B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 11</td>
<td>ST4B (scalar plus scalar)</td>
</tr>
<tr>
<td>01 01</td>
<td>ST2H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 10</td>
<td>ST3H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 11</td>
<td>ST4H (scalar plus scalar)</td>
</tr>
<tr>
<td>10 01</td>
<td>ST2W (scalar plus scalar)</td>
</tr>
<tr>
<td>10 10</td>
<td>ST3W (scalar plus scalar)</td>
</tr>
<tr>
<td>10 11</td>
<td>ST4W (scalar plus scalar)</td>
</tr>
<tr>
<td>11 01</td>
<td>ST2D (scalar plus scalar)</td>
</tr>
<tr>
<td>11 10</td>
<td>ST3D (scalar plus scalar)</td>
</tr>
<tr>
<td>11 11</td>
<td>ST4D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE Memory - Scatter with Optional Sign Extend

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)</td>
</tr>
</tbody>
</table>
SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE Memory - Scatter

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 64-bit scatter store (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (vector plus immediate)</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

These instructions are under SVE Memory - Scatter.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

These instructions are under SVE Memory - Scatter.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (vector plus immediate)

These instructions are under SVE Memory - Scatter.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (vector plus immediate)</td>
</tr>
</tbody>
</table>
SVE 32-bit scatter store (vector plus immediate)

These instructions are under SVE Memory - Scatter.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Memory - Contiguous Store with Immediate Offset

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE contiguous non-temporal store (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>SVE store multiple structures (scalar plus immediate)</td>
</tr>
<tr>
<td>!= 00</td>
<td>SVE contiguous store (scalar plus immediate)</td>
</tr>
<tr>
<td>0</td>
<td>SVE contiguous non-temporal store (scalar plus immediate)</td>
</tr>
</tbody>
</table>

SVE contiguous non-temporal store (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Store with Immediate Offset.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

SVE store multiple structures (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Store with Immediate Offset.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00 01</td>
<td>ST3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00 11</td>
<td>ST4B (scalar plus immediate)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00
### Decode fields

#### msz

| 01 | 01 | ST2H (scalar plus immediate) |
| 01 | 10 | ST3H (scalar plus immediate) |
| 01 | 11 | ST4H (scalar plus immediate) |
| 10 | 01 | ST2W (scalar plus immediate) |
| 10 | 10 | ST3W (scalar plus immediate) |
| 10 | 11 | ST4W (scalar plus immediate) |
| 11 | 01 | ST2D (scalar plus immediate) |
| 11 | 10 | ST3D (scalar plus immediate) |
| 11 | 11 | ST4D (scalar plus immediate) |

#### SVE contiguous store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

### Data Processing -- Immediate

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 msz size 0 imm4 1 1 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

### Instruction Details

#### msz

| 00 | ST1B (scalar plus immediate) |
| 01 | ST1H (scalar plus immediate) |
| 10 | ST1W (scalar plus immediate) |
| 11 | ST1D (scalar plus immediate) |

### PC-rel. addressing

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>op</th>
<th>immlo</th>
<th>immhi</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>immlo</td>
<td>1 0 0 0 0</td>
<td>immhi</td>
<td>Rd</td>
</tr>
</tbody>
</table>

### Instruction Details

#### op0

| 00x | PC-rel. addressing |
| 010 | Add/subtract (immediate) |
| 011 | Add/subtract (immediate, with tags) |
| 100 | Logical (immediate) |
| 101 | Move wide (immediate) |
| 110 | Bitfield |
| 111 | Extract |

#### op

| 0 | ADR |
| 1 | ADRP |
Add/subtract (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Instruction Details**

- **ADD (immediate)** — 32-bit
- **ADDS (immediate)** — 32-bit
- **SUB (immediate)** — 32-bit
- **SUBS (immediate)** — 32-bit
- **ADD (immediate)** — 64-bit
- **ADDS (immediate)** — 64-bit
- **SUB (immediate)** — 64-bit
- **SUBS (immediate)** — 64-bit

Add/subtract (immediate, with tags)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>imm6</th>
<th>op3</th>
<th>imm4</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Instruction Details**

- **ADDG**
- **SUBG**

**Feature**

- **FEAT_MTE**

Logical (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Instruction Details**

- **AND (immediate)** — 32-bit
- **ORR (immediate)** — 32-bit
- **EOR (immediate)** — 32-bit
- **ANDS (immediate)** — 32-bit
- **AND (immediate)** — 64-bit
- **ORR (immediate)** — 64-bit
- **EOR (immediate)** — 64-bit
- **ANDS (immediate)** — 64-bit

Move wide (immediate)

These instructions are under **Data Processing -- Immediate**.
Top-level encodings for A64

### Bitfield

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>OPC</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>00 0x</td>
</tr>
<tr>
<td>0</td>
<td>10 0x</td>
</tr>
<tr>
<td>0</td>
<td>11 0x</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

### Extract

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>OPC</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>00 0</td>
</tr>
<tr>
<td>0</td>
<td>01 0</td>
</tr>
<tr>
<td>0</td>
<td>10 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00 1</td>
</tr>
<tr>
<td>1</td>
<td>01 1</td>
</tr>
<tr>
<td>1</td>
<td>10 1</td>
</tr>
</tbody>
</table>

### Branches, Exception Generating and System instructions

These instructions are under the **top-level**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>OP21</td>
</tr>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>00 0 0 0 0xxxxx</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00 1 0</td>
</tr>
</tbody>
</table>
### Conditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>0xxxxxxxxxxxxx</td>
<td></td>
<td>Conditional branch (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>00xxxxxxxxxxxx</td>
<td></td>
<td>Exception generation</td>
</tr>
<tr>
<td>110</td>
<td>01000000110001</td>
<td></td>
<td>System instructions with register argument</td>
</tr>
<tr>
<td>110</td>
<td>01000001100110 1111</td>
<td></td>
<td>Hints</td>
</tr>
<tr>
<td>110</td>
<td>01000001100111</td>
<td></td>
<td>Barriers</td>
</tr>
<tr>
<td>110</td>
<td>01000000xxxx0100</td>
<td></td>
<td>PSTATE</td>
</tr>
<tr>
<td>110</td>
<td>0100x01xxxxxxxx</td>
<td></td>
<td>System instructions</td>
</tr>
<tr>
<td>110</td>
<td>0100x1xxxxxxx</td>
<td></td>
<td>System register move</td>
</tr>
<tr>
<td>110</td>
<td>1xxxxxxxxxxxxxx</td>
<td></td>
<td>Unconditional branch (register)</td>
</tr>
<tr>
<td>x00</td>
<td>0xxxxxxxxxxxxx</td>
<td></td>
<td>Unconditional branch (immediate)</td>
</tr>
<tr>
<td>x01</td>
<td>0xxxxxxxxxxxxx</td>
<td></td>
<td>Compare and branch (immediate)</td>
</tr>
<tr>
<td>x01</td>
<td>1xxxxxxxxxxxxx</td>
<td></td>
<td>Test and branch (immediate)</td>
</tr>
</tbody>
</table>

### Exception generation

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 0 0</td>
<td>B.cond</td>
<td>-</td>
</tr>
<tr>
<td>o1 0 1</td>
<td>BC.cond</td>
<td>FEAT_HBC</td>
</tr>
<tr>
<td>1 0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Top-level encodings for A64
## System instructions with register argument

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRm op2 LL</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0000 000 1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>100 000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>101 000 00</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>101 000 01</td>
<td>DCPS1</td>
<td></td>
</tr>
<tr>
<td>101 000 10</td>
<td>DCPS2</td>
<td></td>
</tr>
<tr>
<td>101 000 11</td>
<td>DCPS3</td>
<td></td>
</tr>
<tr>
<td>110 000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>111 000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

## Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRm op2 CRm</td>
<td>HINT</td>
<td></td>
</tr>
<tr>
<td>0000 000</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>0000 001</td>
<td>YIELD</td>
<td></td>
</tr>
<tr>
<td>0000 010</td>
<td>WFE</td>
<td></td>
</tr>
<tr>
<td>0000 011</td>
<td>WFI</td>
<td></td>
</tr>
<tr>
<td>0000 100</td>
<td>SEV</td>
<td></td>
</tr>
<tr>
<td>0000 101</td>
<td>SEVL</td>
<td></td>
</tr>
<tr>
<td>0000 110</td>
<td>DGH</td>
<td></td>
</tr>
<tr>
<td>0000 111</td>
<td>XPACD, XPACI, XPACLRI</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0001 000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0001 010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0001 100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0001 110</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0010 000</td>
<td>ESB</td>
<td>FEAT_RAS</td>
</tr>
<tr>
<td>0010 001</td>
<td>PSB.CSYNC</td>
<td>FEAT_SPE</td>
</tr>
<tr>
<td>0010 010</td>
<td>TSB.CSYNC</td>
<td>FEAT_TRF</td>
</tr>
<tr>
<td>0010 100</td>
<td>CSDR</td>
<td></td>
</tr>
<tr>
<td>0011 000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ</td>
<td>FEAT_PAuth</td>
</tr>
</tbody>
</table>
### Barriers

These instructions are under **Branches, Exception Generating and System instructions**.

<table>
<thead>
<tr>
<th>CRm</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>op2 001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>op2 010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>op2 011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>DSB — memory barrier</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>11111</td>
<td>DSB</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>11111</td>
<td>ISB</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>11111</td>
<td>SB</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>11111</td>
<td>SB</td>
<td>-</td>
</tr>
<tr>
<td>xx0x</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>xx10</td>
<td>001</td>
<td>DSB — Memory nXS barrier</td>
<td>FEAT_XS</td>
</tr>
</tbody>
</table>

### PSTATE

These instructions are under **Branches, Exception Generating and System instructions**.

<table>
<thead>
<tr>
<th>CRm</th>
<th>op1</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>11111</td>
<td>MSR (immediate)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>CFINV</td>
<td>FEAT_FlagM</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>XAFLAG</td>
<td>FEAT_FlagM2</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>AXFLAG</td>
<td>FEAT_FlagM2</td>
<td></td>
</tr>
</tbody>
</table>
System instructions

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>L</th>
<th>0</th>
<th>SYS</th>
<th>1</th>
<th>SYSL</th>
</tr>
</thead>
</table>

System register move

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>L</th>
<th>0</th>
<th>MSR (register)</th>
<th>1</th>
<th>MRS</th>
</tr>
</thead>
</table>

Unconditional branch (register)

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>Decode fields</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>!11111</td>
<td>0000000</td>
<td>!0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 0000000</td>
<td>!0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 0000000</td>
<td>00000</td>
<td>BR</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 000001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 000010</td>
<td>!11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 000010</td>
<td>11111</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 000011</td>
<td>!11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 000011</td>
<td>11111</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 11111 1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 11111 000000</td>
<td>!0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 11111 000000</td>
<td>00000</td>
<td>BLR</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 11111 000001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>op2</td>
<td>Decode fields</td>
<td>op4</td>
<td>Instruction Details</td>
<td>Feature</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>---------------</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>000010</td>
<td>!1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>000011</td>
<td>!1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>000011</td>
<td>11111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000000</td>
<td>!0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000000</td>
<td>00000</td>
<td>RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>!1111 !1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>!1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000010</td>
<td>11111</td>
<td>REETA, RETAB — RETAA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000011</td>
<td>!1111 !1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000011</td>
<td>!1111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000011</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000011</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>000011</td>
<td>11111</td>
<td>REETA, RETAB — RETAB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>001xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>1111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>1111</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>!1111 !0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>!1111 000000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>11111 !00000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>11111 !0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>11111 !00000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>000000</td>
<td>11111 !0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unconditional branch (immediate)

These instructions are under **Branches, Exception Generating and System instructions**.
### Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>BBL</td>
</tr>
</tbody>
</table>

### Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>b5</th>
<th>op</th>
<th>imm14</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CBZ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CBNZ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CBZ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CBNZ</td>
</tr>
</tbody>
</table>

### Loads and Stores

These instructions are under the [top-level](#).

<table>
<thead>
<tr>
<th>op</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1xxx</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0xxxxx</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0xxxxx</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000000</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>x00000</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1xxxxx</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xx1xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xxx1xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xxxx1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xxxxx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>1x</td>
<td>1xxx</td>
<td>Load/store memory tags</td>
</tr>
<tr>
<td>0x00</td>
<td>0</td>
<td>00</td>
<td>1xxxxx</td>
<td>Load/store exclusive pair</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
Load/store exclusive register
Load/store ordered
Compare and swap
LDAPR/STLR (unscaled immediate)
Load register (literal)
Memory Copy and Memory Set
Load/store no-allocate pair (offset)
Load/store register pair (post-indexed)
Load/store register pair (offset)
Load/store register pair (pre-indexed)
Load/store register (unscaled immediate)
Load/store register (immediate post-indexed)
Load/store register (unprivileged)
Load/store register (immediate pre-indexed)
Load/store register (register offset)
Load/store register (pac)
Load/store register (unsigned immediate)

Compare and swap pair

These instructions are under Loads and Stores.

Advanced SIMD load/store multiple structures

These instructions are under Loads and Stores.
## Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>L</th>
<th>Decode fields Rm</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>ST2 (multiple structures)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>ST1 (multiple structures) — two registers</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>11xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>LD4 (multiple structures)</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>LD3 (multiple structures)</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>LD1 (multiple structures) — one register</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>LD2 (multiple structures)</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>11xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### Advanced SIMD load/store single structure

These instructions are under [Loads and Stores](#).

---

<table>
<thead>
<tr>
<th>L</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111</td>
<td>1000</td>
<td>ST2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>1010</td>
<td>ST1 (multiple structures) — two registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>11xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0000</td>
<td>LD4 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0100</td>
<td>LD3 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0111</td>
<td>LD1 (multiple structures) — one register, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>1000</td>
<td>LD2 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0000</td>
<td>LD4 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0100</td>
<td>LD3 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>1000</td>
<td>LD2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, immediate offset</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>L</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td></td>
<td>ST1 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td></td>
<td>ST3 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>x0</td>
<td>ST1 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>x0</td>
<td>ST3 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>00</td>
<td>ST1 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>00</td>
<td>ST3 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>1</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>00</td>
<td>ST2 (single structure) — 8-bit</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Decode fields</td>
<td>S</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------------</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>10</td>
</tr>
</tbody>
</table>
## Advanced SIMD load/store single structure (post-indexed)

These instructions are under [Loads and Stores](#).

### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>101</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>0</td>
<td></td>
<td>LD2R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>0</td>
<td></td>
<td>LD4R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>101</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>0</td>
<td>00</td>
<td>ST1 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>001</td>
<td></td>
<td>ST3 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td>ST1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td>ST3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td>ST1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>01</td>
<td>ST1 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>ST3 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>01</td>
<td>ST3 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>0001</td>
<td></td>
<td>ST1 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>010 x0</td>
<td></td>
<td>ST1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>011 x0</td>
<td></td>
<td>ST3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>ST1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>ST1 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>ST3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>ST3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Rm</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>--------</td>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td>ST2 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>001</td>
<td>ST4 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td>ST2 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td>ST4 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td>ST2 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>01</td>
<td>ST2 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>ST4 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>01</td>
<td>ST4 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td>ST2 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td>ST4 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>ST2 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>ST4 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>ST2 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>ST2 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>ST4 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>ST4 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>ST2 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>ST4 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>ST2 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>ST4 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>LD1 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>000</td>
<td>LD1 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>001</td>
<td>LD3 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD3 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### Load/store memory tags

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>imm9</th>
<th>op2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>STG — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>STG — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>STG — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0000000000</td>
<td>STZGM</td>
<td>FEAT_MTE2</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LDG</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>STZG — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>STZG — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>STZG — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>ST2G — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>ST2G — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>ST2G — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>Decode fields</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>10</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0 11</td>
<td>11</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>1  x1</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>101</td>
<td>10</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0 11</td>
<td>11</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>1  x1</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>110</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>111</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>001</td>
<td>LD4 (single structure) — 8-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>010</td>
<td>LD2 (single structure) — 16-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>011</td>
<td>LD4 (single structure) — 16-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>100</td>
<td>LD2 (single structure) — 32-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>101</td>
<td>LD4 (single structure) — 32-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>110</td>
<td>LD2R — register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>111</td>
<td>LD4R — register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>001</td>
<td>LD4 (single structure) — 8-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>010</td>
<td>LD2 (single structure) — 16-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>011</td>
<td>LD4 (single structure) — 16-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>100</td>
<td>LD2 (single structure) — 32-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>101</td>
<td>LD4 (single structure) — 32-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>110</td>
<td>LD2R — immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>111</td>
<td>LD4R — immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Table of Instructions**

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>Decode fields</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>10</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0 11</td>
<td>11</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>1  x1</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>101</td>
<td>10</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0 11</td>
<td>11</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>1  x1</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>110</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>111</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>001</td>
<td>LD4 (single structure) — 8-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>010</td>
<td>LD2 (single structure) — 16-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>011</td>
<td>LD4 (single structure) — 16-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>100</td>
<td>LD2 (single structure) — 32-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>101</td>
<td>LD4 (single structure) — 32-bit, register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>110</td>
<td>LD2R — register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 1111</td>
<td>111</td>
<td>LD4R — register offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>001</td>
<td>LD4 (single structure) — 8-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>010</td>
<td>LD2 (single structure) — 16-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>011</td>
<td>LD4 (single structure) — 16-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>100</td>
<td>LD2 (single structure) — 32-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>101</td>
<td>LD4 (single structure) — 32-bit, immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>110</td>
<td>LD2R — immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111</td>
<td>111</td>
<td>LD4R — immediate offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Top-level encodings for A64**

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>Decode fields</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>01</td>
<td>STG — post-index</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>10</td>
<td>STG — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
<td>STG — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0000000000</td>
<td>00</td>
<td>STZGM</td>
<td>FEAT_MTE2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>LDG</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>STZG — post-index</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>10</td>
<td>STZG — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>11</td>
<td>STZG — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>01</td>
<td>ST2G — post-index</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>ST2G — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>ST2G — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Load/store exclusive pair

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>!= 0000000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0000000000</td>
<td>STGM</td>
<td>FEAT MTE2</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>STZ2G — post-index</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>STZ2G — signed offset</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>STZ2G — pre-index</td>
<td>FEAT MTE</td>
</tr>
<tr>
<td>11</td>
<td>!= 0000000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0000000000</td>
<td>LDGM</td>
<td>FEAT MTE2</td>
</tr>
</tbody>
</table>

### Load/store exclusive register

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>sz L o0</td>
<td>STXP — 32-bit</td>
</tr>
<tr>
<td>0 0 0</td>
<td>STLXP — 32-bit</td>
</tr>
<tr>
<td>0 1 0</td>
<td>LDXP — 32-bit</td>
</tr>
<tr>
<td>0 1 1</td>
<td>LDAXP — 32-bit</td>
</tr>
<tr>
<td>1 0 0</td>
<td>STXP — 64-bit</td>
</tr>
<tr>
<td>1 0 1</td>
<td>STLXP — 64-bit</td>
</tr>
<tr>
<td>1 1 0</td>
<td>LDXP — 64-bit</td>
</tr>
<tr>
<td>1 1 1</td>
<td>LDAXP — 64-bit</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>o0</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>LDAXR — 64-bit</td>
</tr>
</tbody>
</table>

### Load/store ordered

These instructions are under [Loads and Stores](#).

### Compare and swap

These instructions are under [Loads and Stores](#).
### LDAPR/STLR (unscaled immediate)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0 0</td>
<td>1111</td>
<td>CAS, CASA, CASAL, CASL — 64-bit CAS</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1</td>
<td>1111</td>
<td>CAS, CASA, CASAL, CASL — 64-bit CASL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>1 0</td>
<td>1111</td>
<td>CAS, CASA, CASAL, CASL — 64-bit CASA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>1 1</td>
<td>1111</td>
<td>CAS, CASA, CASAL, CASL — 64-bit CASAL</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>

### Load register (literal)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>STLURB</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00 01</td>
<td>LDAPURB</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00 10</td>
<td>LDAPURSB — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>00 11</td>
<td>LDAPURSB — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 00</td>
<td>STLURH</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 01</td>
<td>LDAPURH</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 10</td>
<td>LDAPURSH — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>01 11</td>
<td>LDAPURSH — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 00</td>
<td>STLR — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 01</td>
<td>LDAPUR — 32-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 10</td>
<td>LDAPURSW</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>10 11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11 00</td>
<td>STLR — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>11 01</td>
<td>LDAPUR — 64-bit</td>
<td>FEAT_LRCPC2</td>
</tr>
<tr>
<td>11 10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11 11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
### Memory Copy and Memory Set

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 0000</td>
<td>CPYFP, CPYFM, CPYFE — CPYFP</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0001</td>
<td>CPYFPWT, CPYFMWT, CPYFEWT — CPYFPWT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0010</td>
<td>CPYFPRT, CPYFMRT, CPYFERT — CPYFPRT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0011</td>
<td>CPYFPFT, CPYFMT, CPYFET — CPYFPFT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0100</td>
<td>CPYFPWN, CPYFMWN, CPYFEWN — CPYFPWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0101</td>
<td>CPYFPWWTN, CPYFMWTWN, CPYFEWTWN — CPYFPWTWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0110</td>
<td>CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFPRTWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 0111</td>
<td>CPYFPFTWN, CPYFMTWN, CPYFETWN — CPYFPFTWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1000</td>
<td>CPYFPN, CPYFMN, CPYFEN — CPYFPN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 1001</td>
<td>CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFPWTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 1010</td>
<td>CPYFPRTTRN, CPYFMRTTRN, CPYFERTTRN — CPYFPRTTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 00 1101</td>
<td>CPYFPFTTRN, CPYFMTFTTRN, CPYFETFTTRN — CPYFPFTTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 0000</td>
<td>CPYFP, CPYFM, CPYFE — CPYFM</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 0001</td>
<td>CPYFPWT, CPYFMWT, CPYFEWT — CPYFMWT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 0010</td>
<td>CPYFPRT, CPYFMRT, CPYFERT — CPYFMRT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 0011</td>
<td>CPYFPFT, CPYFMT, CPYFET — CPYFMT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1000</td>
<td>CPYFPWN, CPYFMWN, CPYFEWN — CPYFMWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1001</td>
<td>CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFMWTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1010</td>
<td>CPYFPRTTRN, CPYFMRTTRN, CPYFERTTRN — CPYFMRTTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1101</td>
<td>CPYFPFTTRN, CPYFMTFTTRN, CPYFETFTTRN — CPYFPFTTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1110</td>
<td>CPYFPN, CPYFMN, CPYFEN — CPYFMN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 01 1111</td>
<td>CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFMWTRN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 10 1000</td>
<td>CPYFP, CPYFM, CPYFE — CPYFE</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 10 0000</td>
<td>CPYFPWT, CPYFMWT, CPYFEWT — CPYFMWT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 10 0001</td>
<td>CPYFPRT, CPYFMRT, CPYFERT — CPYFERT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 10 0010</td>
<td>CPYFPN, CPYFMN, CPYFEN — CPYFEN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>00 10 1000</td>
<td>CPYFPN, CPYFMN, CPYFEN — CPYFEN</td>
<td>FEAT_MOPS</td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 10 1010</td>
<td>CPYFPTRN, CPYFMRTRN, CPYFERTRN — CPYFERTRN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 10 1011</td>
<td>CPYFPTRN, CPYFMRTRN, CPYFERTRN — CPYFERTRN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 10 1100</td>
<td>CPYFPN, CPYFMN, CPYFE — CPYFN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 10 1101</td>
<td>CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFEWTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 10 1110</td>
<td>CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFERTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 10 1111</td>
<td>CPYFPTN, CPYFMTN, CPYFETN — CPYFETN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0000</td>
<td>SETP, SETM, SETE — SETP</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0001</td>
<td>SETPT, SETMT, SETET — SETPT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0010</td>
<td>SETPN, SETMN, SETEN — SETPN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0011</td>
<td>SETPTN, SETMTN, SETETN — SETPTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0100</td>
<td>SETP, SETM, SETE — SETM</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0101</td>
<td>SETPT, SETMT, SETET — SETMT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0110</td>
<td>SETPN, SETMN, SETEN — SETMN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 0111</td>
<td>SETPTN, SETMTN, SETETN — SETMTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 1000</td>
<td>SETP, SETM, SETE — SETE</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 1001</td>
<td>SETPT, SETMT, SETET — SETET</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 1010</td>
<td>SETPN, SETMN, SETEN — SETEN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 1011</td>
<td>SETPTN, SETMTN, SETETN — SETETN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>0 11 11xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 00 0000</td>
<td>CPYPW, CPYM, CPYE — CPYP</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0001</td>
<td>CPYPWT, CPYMWT, CPYEWT — CPYWT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0010</td>
<td>CPYPRRT, CPYMRRT, CPYERT — CPYRT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0011</td>
<td>CPYPT, CPYMT, CPYET — CPYPT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0100</td>
<td>CPYPWN, CPYMWN, CPYEWN — CPYPWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0101</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWTWN — CPYWTWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0110</td>
<td>CPYPRTWN, CPYMRWTN, CPYERTWN — CPYRTWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 0111</td>
<td>CPYPTWN, CPYMTWN, CPYETWN — CPYWTWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1000</td>
<td>CPYPN, CPYMN, CPYEN — CPYN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1001</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN — CPYWTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1010</td>
<td>CPYPRTN, CPYMRTN, CPYERTN — CPYRTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1011</td>
<td>CPYPTRN, CPYMRTRN, CPYERTN — CPYERTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1100</td>
<td>CPYPN, CPYMN, CPYEN — CPYN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1101</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN — CPYWTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1110</td>
<td>CPYPRTN, CPYMRTRN, CPYERTN — CPYRTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 00 1111</td>
<td>CPYPTN, CPYMTN, CPYETN — CPYPTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0000</td>
<td>CPYPW, CPYM, CPYE — CPYP</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0001</td>
<td>CPYPWT, CPYMWT, CPYEWT — CPYWT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0010</td>
<td>CPYPRRT, CPYMRRT, CPYERT — CPYRT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0011</td>
<td>CPYPT, CPYMT, CPYET — CPYPT</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0100</td>
<td>CPYPWN, CPYMWN, CPYEWN — CPYPWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0101</td>
<td>CPYPWTWN, CPYMWTWN, CPYEWTWN — CPYWTWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0110</td>
<td>CPYPRTWN, CPYMRWTN, CPYERTWN — CPYRTWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 0111</td>
<td>CPYPTWN, CPYMTWN, CPYETWN — CPYWTWN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 1000</td>
<td>CPYPN, CPYMN, CPYEN — CPYN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 1001</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN — CPYWTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 1010</td>
<td>CPYPRTN, CPYMRTRN, CPYERTN — CPYRTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>1 01 1011</td>
<td>CPYPTN, CPYMTN, CPYETN — CPYPTN</td>
<td>FEAT MOPS</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
<td>Feature</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>1 01 1100</td>
<td>CPYPN, CPYMN, CPYEN — CPYMN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 1101</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN — CPYMWTN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 01 1110</td>
<td>CPYRPN, CPYMRTN, CPYERTN — CPYRTN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0000</td>
<td>CPYP, CPYM, CPYE — CPYE</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0001</td>
<td>CPYPWT, CPYMWT, CPYEWT — CPYEWT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0010</td>
<td>CPYPRPT, CPYMRTRN, CPYERT — CPYERT</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0101</td>
<td>CPYPTWN, CPYMWTWN, CPYEWTWN — CPYEWTWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0110</td>
<td>CPYPRTN, CPYMRTWN, CPYERTWN — CPYERTWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 0111</td>
<td>CPYPTWN, CPYMTRWN, CPYETWN — CPYETWN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1000</td>
<td>CPYPRTN, CPYMRTN, CPYEWN — CPYRTN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 10 1001</td>
<td>CPYPWTN, CPYMWTN, CPYEWTN — CPYWTN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0000</td>
<td>SETGP, SETGM, SETGE — SETGP</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0001</td>
<td>SETGP, SETGMT, SETGET — SETGP</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0010</td>
<td>SETGPN, SETGMN, SETGEN — SETGPN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0011</td>
<td>SETGPTN, SETGMTN, SETGETN — SETGPTN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0100</td>
<td>SETGP, SETGM, SETGE — SETGM</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0101</td>
<td>SETGPT, SETGMT, SETGET — SETGM</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0110</td>
<td>SETGPN, SETGMN, SETGEN — SETGMN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 0111</td>
<td>SETGPTN, SETGMTN, SETGETN — SETGTMN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 1000</td>
<td>SETGP, SETGM, SETGE — SETGE</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 1001</td>
<td>SETGPT, SETGMT, SETGET — SETGET</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 1010</td>
<td>SETGPN, SETGMN, SETGEN — SETGEN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 1011</td>
<td>SETGPTN, SETGMTN, SETGETN — SETGETN</td>
<td>FEAT_MOPS</td>
</tr>
<tr>
<td>1 11 11xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

**Load/store no-allocate pair (offset)**

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>STNP — 32-bit</td>
</tr>
<tr>
<td>00 0 1</td>
<td>LDNP — 32-bit</td>
</tr>
<tr>
<td>00 1 0</td>
<td>STNP (SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>00 1 1</td>
<td>LDNP (SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>01 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 1 0</td>
<td>STNP (SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>
### Load/store register pair (post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STNP — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDNP — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Load/store register pair (offset)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STGP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDPSW</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
### Load/store register pair (pre-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Load/store register (unscaled immediate)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Decode fields

- **opc**: Operation code
- **V**: Immediate size
- **L**: Length of the register pair
- **imm7**: Immediate value for the load/store instruction
- **Rt2**: Destination register
- **Rn**: Source register
- **Rt**: Third register

- **STP**: Store register pair
- **LDP**: Load register pair
- **STUR**: Store register (unscaled immediate)
- **LDUR**: Load register (unscaled immediate)
- **STURH**: Store register high
- **LDURH**: Load register high

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>
### Decode fields and Instruction Details

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Load/store register (immediate post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STUR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDUR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDURSW</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STUR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDUR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFUM</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>
Load/store register (unprivileged)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>opc</th>
<th>0</th>
<th>imm9</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STTRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDTRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDTRSB — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDTRSB — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STTRH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDTRH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDTRSH — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDTRSH — 32-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STTR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDTR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDTRSW</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Load/store register (immediate pre-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>opc</th>
<th>0</th>
<th>imm9</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

**Decode fields**

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 01</td>
<td>LDR (immediate) — 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 0 10</td>
<td>LDRSW (immediate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 0 00</td>
<td>STR (immediate) — 64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 0 01</td>
<td>LDR (immediate) — 64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 0 10</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Atomic memory operations**

These instructions are under **Loads and Stores**.

---

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V A R Rs o3 opc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 11x UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 100 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 001 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 010 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 011 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 101 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 001 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 010 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 011 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 101 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 001 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 010 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 011 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 100 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 101 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 UNALLOCATED -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 000 LDADDDB, LDADDDB, LDADDLDB, LDADDLB — LDADDR</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 001 LDCLRDB, LDCLRDB, LDCLRDB, LDCLRDB — LDCLRDB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 010 LDEORDB, LDEORDB, LDEORDB, LDEORDB — LDEORDB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 011 LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 100 LDSMAXB, LDSMAXB, LDSMAXALB, LDSMAXLB — LDSMAXB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 101 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 110 LDUMAXB, LDUMAXB, LDUMAXALB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00 0 0 0 0 0 111 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINB</td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Top-level encodings for A64

Page 2685
<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>LDAPRH</td>
<td>FEAT_LRCPC</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDH, LDADDAH, LDADDAHL, LDADDLH — LDADDAHL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRRAH, LDCLRRAHL, LDCLRLH — LDCLRRLH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORALHL — LDEORALHL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETALHL — LDSETALHL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORAL — 32-bit LDEOR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETAL — 32-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDADDL, LDADDA, LDADDA, LDADDL — 32-bit LDADDL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDADDLH, LDADDAHL, LDADDAH, LDADDLHL — 32-bit LDADDLH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Feature</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDMSMAX, LDMSMAXA, LDMSMAXAL, LDMSAXL — 32-bit LDMSAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDMSMIN, LDMSMINA, LDMSMINAL, LDMSMINL — 32-bit LDMSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDMSMAX, LDMSMAXA, LDMSMAXAL, LDMSAXL — 32-bit LDMSAXA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDMSMIN, LDMSMINA, LDMSMINAL, LDMSMINL — 32-bit LDMSMINA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDAPR — 32-bit</td>
<td>FEAT_LRCPC</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDMSMAX, LDMSMAXA, LDMSMAXAL, LDMSAXL — 32-bit LDMSAXAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDMSMIN, LDMSMINA, LDMSMINAL, LDMSMINL — 32-bit LDMSMINAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Feature</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>ST64BV0</td>
<td>FEAT_LS64_ACCDATA</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>ST64BV</td>
<td>FEAT_LS64_V</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>1</td>
<td>001</td>
<td>ST64B</td>
<td>FEAT_LS64</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>1</td>
<td>101</td>
<td>LD64B</td>
<td>FEAT_LS64</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>LDADDA, LDADDA, LDADDA, LDADDA — 64-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>LDADDA, LDADDA, LDADDA, LDADDA — 64-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
Load/store register (register offset)

These instructions are under **Loads and Stores**.
### Load/store register (pac)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>option</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STR (register) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td></td>
<td>LDR (register) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td></td>
<td>PRFM (register)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
</tbody>
</table>

### Load/store register (unsigned immediate)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFM (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

### Data Processing -- Register

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td></td>
<td>Data-processing (2 source)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0110</td>
<td></td>
<td>Data-processing (1 source)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0xxx</td>
<td></td>
<td>Logical (shifted register)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1xx0</td>
<td></td>
<td>Add/subtract (shifted register)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1xx1</td>
<td></td>
<td>Add/subtract (extended register)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>00000</td>
<td>Add/subtract (with carry)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>x00001</td>
<td>Rotate right into flags</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>xx0010</td>
<td>Evaluate into flags</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>xxxx0x</td>
<td>Conditional compare (register)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>xxxx1x</td>
<td>Conditional compare (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0100</td>
<td></td>
<td>Conditional select</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1xxx</td>
<td></td>
<td>Data-processing (3 source)</td>
</tr>
</tbody>
</table>

### Data-processing (2 source)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>opcode</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000011x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data-processing (1 source)

These instructions are under Data Processing -- Register.

### Top-level encodings for A64

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000010</td>
<td>UDIV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000011</td>
<td>SDIV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00010x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>001000</td>
<td>LSLV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>001001</td>
<td>LSRV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>001010</td>
<td>ASRV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>001011</td>
<td>RORV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010x11</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010000</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32B</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010001</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32H</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010010</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32W</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010100</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010101</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>010110</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000000</td>
<td>SUBP</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000010</td>
<td>UDIV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000011</td>
<td>SDIV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000100</td>
<td>IRG</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000101</td>
<td>GMI</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001000</td>
<td>LSLV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001001</td>
<td>LSRV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001010</td>
<td>ASRV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001011</td>
<td>RORV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001100</td>
<td>PACGA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>010xx0</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>010x0x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>010011</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>010111</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>100000</td>
<td>SUBPS</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode2</th>
<th>opcode</th>
<th>Rn</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
<td>0011x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000000</td>
<td>001xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Page 2692
<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Rn</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000000</td>
<td>RBIT — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000001</td>
<td>REV16 — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000010</td>
<td>REV — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000100</td>
<td>CLZ — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000</td>
<td>000101</td>
<td>CLS — 32-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000000</td>
<td>RBIT — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000001</td>
<td>REV16 — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000010</td>
<td>REV32</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000011</td>
<td>REV — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000100</td>
<td>CLZ — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000101</td>
<td>CLS — 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000001</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000010</td>
<td>PACDA, PACDZA — PACDA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000011</td>
<td>PACDB, PACDZB — PACDB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000101</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000110</td>
<td>AUTDA, AUTDZA — AUTDA</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>000111</td>
<td>AUTDB, AUTDZB — AUTDB</td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001000</td>
<td>11111</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001001</td>
<td>11111</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001010</td>
<td>11111</td>
<td>PACDA, PACDZA — PACDZA</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001011</td>
<td>11111</td>
<td>PACDB, PACDZB — PACDZB</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001100</td>
<td>11111</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001101</td>
<td>11111</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001110</td>
<td>11111</td>
<td>AUTDA, AUTDZA — AUTDA</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>001111</td>
<td>11111</td>
<td>AUTDB, AUTDZB — AUTDB</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>010000</td>
<td>11111</td>
<td>XPACD, XPACI, XPACLRI — XPACI</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>010001</td>
<td>11111</td>
<td>XPACD, XPACI, XPACLRI — XPACD</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>010010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

**Logical (shifted register)**

These instructions are under Data Processing -- Register.
### Add/subtract (shifted register)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td></td>
<td>AND (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td>ORR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td></td>
<td>ORN (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td>EOR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td></td>
<td>EON (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td>ANDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td></td>
<td>BICS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td></td>
<td>AND (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
<td>BIC (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td>ORR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td></td>
<td>ORN (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td>EOR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td>EON (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td>ANDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td></td>
<td>BICS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>

### Add/subtract (extended register)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>S</th>
<th>shift</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td></td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td>ADD (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ADDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>SUB (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td>SUBS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td>ADD (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td>ADDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>SUB (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td>SUBS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>SUBS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td>SUBS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>opt</th>
<th>imm3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td></td>
<td>ADD (extended register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td></td>
<td>SUB (extended register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td></td>
<td>SUBS (extended register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
<td>ADD (extended register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td></td>
<td>ADDS (extended register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td></td>
<td>SUB (extended register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td></td>
<td>SUBS (extended register) — 64-bit</td>
</tr>
</tbody>
</table>

### Add/subtract (with carry)

These instructions are under **Data Processing -- Register**.

### Rotate right into flags

These instructions are under **Data Processing -- Register**.

### Evaluate into flags

These instructions are under **Data Processing -- Register**.
### Conditional compare (register)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode2</th>
<th>sz</th>
<th>o3</th>
<th>mask</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!= 000000</td>
<td>0</td>
<td>0</td>
<td>! = 1101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>! = 000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SETF8, SETF16, SETF8 FEAT FlagM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SETF8, SETF16</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SETF8, SETF16, SETF16 FEAT FlagM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Conditional compare (immediate)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode2</th>
<th>sz</th>
<th>o3</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>! = 000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>!= 000000</td>
<td>0</td>
<td>0</td>
<td>! = 1101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SETF8, SETF16, SETF8 FEAT FlagM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SETF8, SETF16</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SETF8, SETF16, SETF16 FEAT FlagM</td>
<td></td>
</tr>
</tbody>
</table>

### Conditional select

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode2</th>
<th>sz</th>
<th>o3</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>! = 000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>!= 000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SETF8, SETF16, SETF8 FEAT FlagM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SETF8, SETF16</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SETF8, SETF16, SETF16 FEAT FlagM</td>
<td></td>
</tr>
</tbody>
</table>
### Data-processing (3 source)

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>CSEL — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>CSINC — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>CSINV — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>CSNEG — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>CSEL — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>CSINC — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>CSINV — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>CSNEG — 64-bit</td>
</tr>
</tbody>
</table>

### Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the [top-level](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op54</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op31</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>op54</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>op31</th>
<th>Rm</th>
<th>o0</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Top-level encodings for A64

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td><strong>Cryptographic AES</strong></td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td><strong>Cryptographic three-register SHA</strong></td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td><strong>Cryptographic two-register SHA</strong></td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>00</td>
<td>00xx</td>
<td>xxx0xxxx1</td>
<td><strong>Advanced SIMD scalar copy</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>01</td>
<td>00xx</td>
<td>xxx0xxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>01</td>
<td>0111</td>
<td>00xxxxx10</td>
<td><strong>Advanced SIMD scalar two-register</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>miscellaneous FP16</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxx0xxxx1</td>
<td><strong>Advanced SIMD scalar three same FP16</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxx01xxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td><strong>Advanced SIMD scalar two-register</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>miscellaneous</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxx1</td>
<td><strong>Advanced SIMD scalar three same extra</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxxx10</td>
<td><strong>Advanced SIMD scalar pairwise</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x1xxxxxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x2xxxxxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x2xxxxxxx1</td>
<td><strong>Advanced SIMD scalar three different</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x3xxxxxxx1</td>
<td><strong>Advanced SIMD scalar three same</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>10</td>
<td>0000</td>
<td>xxx0xxxxx1</td>
<td><strong>Advanced SIMD scalar shift by immediate</strong></td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>11</td>
<td>0000</td>
<td>xxx0xxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>1x</td>
<td>xxx0xxxxx0</td>
<td><strong>Advanced SIMD scalar x indexed element</strong></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td><strong>Advanced SIMD table lookup</strong></td>
<td>-</td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td><strong>Advanced SIMD permute</strong></td>
<td>-</td>
</tr>
<tr>
<td>0x10</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td><strong>Advanced SIMD extract</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>00</td>
<td>00xx</td>
<td>xxx0xxxxx1</td>
<td><strong>Advanced SIMD copy</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>01</td>
<td>00xx</td>
<td>xxx0xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>10xx</td>
<td>xxx01xxx1</td>
<td><strong>Advanced SIMD three same (FP16)</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>10xx</td>
<td>xxx01xxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td><strong>Advanced SIMD two-register miscellaneous (FP16)</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxx1</td>
<td><strong>Advanced SIMD three-register extension</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x1xx</td>
<td>x1xxxxxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx1xxxx0</td>
<td><strong>Advanced SIMD three different</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x1xx</td>
<td>xxx1xxxxx1</td>
<td><strong>Advanced SIMD three same</strong></td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>10</td>
<td>0000</td>
<td>xxx0xxxxx1</td>
<td><strong>Advanced SIMD modified immediate</strong></td>
<td>-</td>
</tr>
</tbody>
</table>
Cryptographic AES

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>AESE</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>AESD</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>AESMC</td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>AESIMC</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Cryptographic three-register SHA

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>SHA1C</td>
</tr>
</tbody>
</table>
### Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>001</td>
<td>SHA1P</td>
</tr>
<tr>
<td>00</td>
<td>010</td>
<td>SHA1M</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>SHA1SU0</td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>SHA256H</td>
</tr>
<tr>
<td>00</td>
<td>101</td>
<td>SHA256H2</td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td>SHA256SU1</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>00000</td>
<td>SHA1H</td>
</tr>
<tr>
<td>00</td>
<td>00001</td>
<td>SHA1SU1</td>
</tr>
<tr>
<td>00</td>
<td>00010</td>
<td>SHA256SU0</td>
</tr>
<tr>
<td>00</td>
<td>00011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar three same FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xxx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xx1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x1xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>DUP (element)</td>
</tr>
<tr>
<td>0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
## Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

### Decode fields

<table>
<thead>
<tr>
<th>Ua</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 1 011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 011</td>
<td>FMULX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 100</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 111</td>
<td>FRECPS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 1 101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 1 111</td>
<td>FRSORTS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 100</td>
<td>FCMGE (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 101</td>
<td>FACGE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 010</td>
<td>FABD</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 100</td>
<td>FCMGT (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 101</td>
<td>FACGT</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Ua</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1100x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 01111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11010</td>
<td>FCVTNS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11011</td>
<td>FCVTMS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11100</td>
<td>FCVTAS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11101</td>
<td>SCVT (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01100</td>
<td>FCMGT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01101</td>
<td>FCMEQ (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01110</td>
<td>FCMLT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11010</td>
<td>FCVTPS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11011</td>
<td>FCVTS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11101</td>
<td>FRECPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11110</td>
<td>FRECPCX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11010</td>
<td>FCVTNU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11011</td>
<td>FCVTMU (vector)</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar three same extra

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>001x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0000</td>
<td>SQRDMLAH (vector)</td>
<td>FEAT RDM</td>
</tr>
<tr>
<td>1 0001</td>
<td>SQRDMLSH (vector)</td>
<td>FEAT RDM</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0010x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x 011xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x 11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 10110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>U</td>
<td>size</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar pairwise**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Advanced SIMD scalar three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 01101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 11011</td>
<td>ADDP (scalar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 01100</td>
<td>FMAXNMP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 01101</td>
<td>FADDP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 01111</td>
<td>FMAXP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 01100</td>
<td>FMINNMP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 01111</td>
<td>FMINP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 11011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0x 01100</td>
<td>FMAXNMP (scalar) — single-precision and double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0x 01101</td>
<td>FADDP (scalar) — single-precision and double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0x 01111</td>
<td>FMAXP (scalar) — single-precision and double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1x 01100</td>
<td>FMINNMP (scalar) — single-precision and double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1x 01111</td>
<td>FMINP (scalar) — single-precision and double-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1001</td>
<td>SQDMLAL, SQDMLAL2 (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1011</td>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1101</td>
<td>SQDMULL, SQDMULL2 (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top-level encodings for A64
<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>011xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>SQADD</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>SQSUB</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>CMGT (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>CMGE (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>SSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>SQSHL (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>SQRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>ADD (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>CMTST</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FMULX</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCMEQ (register)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11111</td>
<td>FRECPS</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>FRSORTS</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>UQSUB</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>CMHI (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>CMHS (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>USHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>UQSHL (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>URSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>UQRSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Decode fields</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>---------------</td>
<td>--------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10110</td>
<td><strong>SQRDMULH (vector)</strong></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10111</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11001</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td><strong>FCMGE (register)</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td><strong>FACGE</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11001</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td><strong>FABD</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td><strong>FCMGT (register)</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td><strong>FACGT</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11110</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td><strong>UNALLOCATED</strong></td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar shift by immediate**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>immh</th>
<th>immb</th>
<th>opcode</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>00000</td>
<td><strong>SSHR</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>00001</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>00010</td>
<td><strong>SSRA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>00011</td>
<td><strong>SRSHR</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>00100</td>
<td><strong>SRSRA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>00101</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01000</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01001</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01010</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01100</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01101</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01110</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>01111</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>10000</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>10001</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>10010</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>10100</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>10101</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11000</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11001</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11010</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11011</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11100</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11101</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11110</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>!= 0000</td>
<td>11111</td>
<td><strong>UNALLOCATED</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

Top-level encodings for A64

---

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10010</td>
<td>SQSHRN, SQSHRN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>10011</td>
<td>SQSHRN, SQSHRN2</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>11100</td>
<td>SCVT (vector, fixed-point)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000</td>
<td>11111</td>
<td>FCVTZS (vector, fixed-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00000</td>
<td>USHR</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00100</td>
<td>USRA</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>00110</td>
<td>URSRA</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01000</td>
<td>SRI</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01010</td>
<td>SLI</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01100</td>
<td>SQSHLU</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>01110</td>
<td>UOUSHL (immediate)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10000</td>
<td>SQSHRUN, SQSHRUN2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10001</td>
<td>SQSHRUN, SQSHRUN2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10010</td>
<td>UOUSHR, UOUSHN2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>10011</td>
<td>UQRSHRN, UQRSHRN2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>11100</td>
<td>UCVTF (vector, fixed-point)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td>11111</td>
<td>FCVTZU (vector, fixed-point)</td>
<td>-</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>opcode</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td></td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td></td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td></td>
<td>SQDMULL, SQDMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td></td>
<td>SQDMULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td></td>
<td>SQRMULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0001</td>
<td>FMLA (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0101</td>
<td>FMLS (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>1001</td>
<td>FMUL (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>0001</td>
<td>FMLA (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>
### Advanced SIMD table lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>UNALLOCATED</td>
<td>TBL — single register table</td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>UNALLOCATED</td>
<td>TBX — single register table</td>
<td></td>
</tr>
<tr>
<td>00000100</td>
<td>UNALLOCATED</td>
<td>TBL — two register table</td>
<td></td>
</tr>
<tr>
<td>00000101</td>
<td>UNALLOCATED</td>
<td>TBX — two register table</td>
<td></td>
</tr>
<tr>
<td>00001000</td>
<td>UNALLOCATED</td>
<td>TBL — three register table</td>
<td></td>
</tr>
<tr>
<td>00001010</td>
<td>UNALLOCATED</td>
<td>TBX — three register table</td>
<td></td>
</tr>
<tr>
<td>00001100</td>
<td>UNALLOCATED</td>
<td>TBL — four register table</td>
<td></td>
</tr>
<tr>
<td>00001110</td>
<td>UNALLOCATED</td>
<td>TBX — four register table</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD permute

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>UZP1</td>
<td></td>
</tr>
<tr>
<td>00000100</td>
<td>TRN1</td>
<td></td>
</tr>
<tr>
<td>00000110</td>
<td>ZIP1</td>
<td></td>
</tr>
<tr>
<td>00001000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00001010</td>
<td>UZP2</td>
<td></td>
</tr>
</tbody>
</table>
## Advanced SIMD extract

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>TRN2</td>
</tr>
<tr>
<td>111</td>
<td>ZIP2</td>
</tr>
</tbody>
</table>

## Advanced SIMD copy

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>EXT</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

## Advanced SIMD three same (FP16)

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>DUP (element)</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>DUP (general)</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>SMOV</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>UMOV</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>INS (general)</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>SMOV</td>
</tr>
<tr>
<td>1</td>
<td>x1000 0111</td>
<td>UMOV</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>INS (element)</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>FMAXNM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>FMLA (vector)</td>
</tr>
</tbody>
</table>
### Advanced SIMD two-register miscellaneous (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

#### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 0 0 010</td>
<td>FADD (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 0 011</td>
<td>FMULX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 0 100</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 0 101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>U 0 0 110</td>
<td>FMAX (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 0 111</td>
<td>FRECPS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 1 000</td>
<td>FMINNM (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 1 001</td>
<td>FMLS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 1 010</td>
<td>FSUB (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 1 011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>U 0 1 100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>U 0 1 101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>U 0 1 110</td>
<td>FMIN (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>U 0 1 111</td>
<td>FRSORTS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 000</td>
<td>FMAXNMP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 010</td>
<td>FADDP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 011</td>
<td>FMUL (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 100</td>
<td>FCMGE (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 101</td>
<td>FACGE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 110</td>
<td>FMAXP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 111</td>
<td>FDIV (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 000</td>
<td>FMINNMP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 010</td>
<td>FABD</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 100</td>
<td>FCMGT (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 101</td>
<td>FACGT</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 110</td>
<td>FMINP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
### Advanced SIMD three-register extension

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

#### Top-level encodings for A64

<table>
<thead>
<tr>
<th>U</th>
<th>a</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11000</td>
<td>FCTAS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11101</td>
<td>SCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01100</td>
<td>FCMGT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01110</td>
<td>FCMLT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01111</td>
<td>FABS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11000</td>
<td>FRINTP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11001</td>
<td>FRINTZ (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11010</td>
<td>FCVTPS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11011</td>
<td>FCVTZS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11100</td>
<td>FRECPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11000</td>
<td>FRINTA (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11001</td>
<td>FRINTX (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
<td>FCMGE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01101</td>
<td>FCMLE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01111</td>
<td>FNEG (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11001</td>
<td>FRINTI (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11101</td>
<td>FRSORTE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>FSORT (vector)</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>

#### Data Processing -- Scalar Floating-Point and Advanced SIMD

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>size</th>
<th>0</th>
<th>Rm</th>
<th>1</th>
<th>opcode</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>Q</th>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>SDOT (vector)</td>
<td>FEAT_DotProd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0011</td>
<td>USDOT (vector)</td>
<td>FEAT_I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>SQRDMLAH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>SQRDMLSH (vector)</td>
<td>FEAT_RDM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Advanced SIMD two-register miscellaneous

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

### Table 1: Decode fields and Instruction Details

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0010</td>
<td>UDOT (vector)</td>
<td>FEAT_DotProd</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10xx</td>
<td>FCMLA</td>
<td>FEAT_FCMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11x0</td>
<td>FCADD</td>
<td>FEAT_FCMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1111</td>
<td>BFDOT (vector)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1111</td>
<td>BFMLALB, BFMLALT (vector)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>0</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>011x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0010</td>
<td>SMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0101</td>
<td>USMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1101</td>
<td>BFMMMLA</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0100</td>
<td>UMMLA (vector)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 2: Decode fields and Instruction Details

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x</td>
<td>011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td>REV64</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>REV16 (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SADDLP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00011</td>
<td>SUQADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>CLS (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>CNT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>SADALP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>SQABS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>CMGT (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>CMEQ (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>CMLT (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>ABS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>XTN, XTN2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Decode size</td>
<td>opcode</td>
<td>Instruction Details</td>
<td>Feature</td>
</tr>
<tr>
<td>---</td>
<td>-------------</td>
<td>--------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01100</td>
<td>SOXTN, SOXTN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10110</td>
<td>FCVTN, FCVTN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10111</td>
<td>FCVTL, FCVTL2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11000</td>
<td>FINTN (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11001</td>
<td>FINTM (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11101</td>
<td>SCVT (vector, integer)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11110</td>
<td>FINT3Z (vector)</td>
<td>FEAT_FINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11111</td>
<td>FINT6Z (vector)</td>
<td>FEAT_FINTTS</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01100</td>
<td>FCMGT (zero)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01110</td>
<td>FCMLT (zero)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01111</td>
<td>FABS (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>FINTP (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>FINTZ (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>FCVTSP (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZS (vector, integer)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11100</td>
<td>URECPE</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>PRECPE</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10110</td>
<td>BFCVTN, BFCVTN2</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>REV32  (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>UADLP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>USQADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>CLZ (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>UADALP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>SQNEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>CMGT (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>CMLE (zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>NEG (vector)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>SQXTUN, SQXTUN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>SHLL, SHLL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UQXN, UQXTN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>10110</td>
<td>FCVTXN, FCVTXN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td>FINTA (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11001</td>
<td>FINTX (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAM (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>UCVTF (vector, integer)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td>FINT32X (vector)</td>
<td>FEAT_FINTTS</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td>FINT64X (vector)</td>
<td>FEAT_FINTTS</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>00101</td>
<td>NOT</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00101</td>
<td>RBIT (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>00101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FCMGE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01101</td>
<td>FCMLE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01111</td>
<td>FNEG (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11001</td>
<td>PRINTI (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td>URSORTE</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FRSORTE</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>FSORT (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0000x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0100x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1100x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>111xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00011</td>
<td>SADDLV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01010</td>
<td>SMAXV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11010</td>
<td>SMINV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11111</td>
<td>ADDV</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01000</td>
<td>FMAXNMV — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01111</td>
<td>FMAX — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01100</td>
<td>FMINNMV — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01111</td>
<td>FMINV — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>UADDLV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UMAXV</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
**Advanced SIMD**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

### Instruction Details

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11010</td>
<td>UMINV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x 01100</td>
<td>FMAXNMV — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x 01111</td>
<td>FMAXV — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x 01100</td>
<td>FMINNMV — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x 01111</td>
<td>FMINV — single-precision and double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

---

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
## Advanced SIMD three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>SHADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>SQADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SRHADD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>SHSUB</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>SQSUB</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>CMGT (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>CMGE (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>SSHL</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>SOSHL (register)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SRSHL</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>SQRSHL</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01100</td>
<td>SMAX</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01101</td>
<td>SMIN</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01110</td>
<td>SABD</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01111</td>
<td>SABA</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>ADD (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>CMTST</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>MLA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>MUL (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SMAXP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10101</td>
<td>SMINP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td>ADDP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11000</td>
<td>FMAXXM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11001</td>
<td>FMLA (vector)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11010</td>
<td>FADD (vector)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11011</td>
<td>FMULX</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11100</td>
<td>FCMEQ (register)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11110</td>
<td>FMAX (vector)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11111</td>
<td>PRECPS</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00011</td>
<td>AND (vector)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>11101</td>
<td>FMLAL, FMLAL2 (vector) — FMLAL</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>00011</td>
<td>BIC (vector, register)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>FMINNM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>FMLS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>ESUB (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11110</td>
<td>FMIN (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>FRSQRTS</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>00011</td>
<td>ORR (vector, register)</td>
</tr>
<tr>
<td>U</td>
<td>Decode fields</td>
<td>size</td>
<td>opcode</td>
</tr>
<tr>
<td>----</td>
<td>---------------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>UHADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>URHADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>UHSUB</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>UQSUB</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>CMHI (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>CMHS (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>USHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>UQSHL (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>URSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>UQRSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01100</td>
<td>UMAX</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01101</td>
<td>UMIN</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01110</td>
<td>UABD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01111</td>
<td>UABA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>MLS (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>PMUL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UMAXP</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UMINP</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10110</td>
<td>SORDMULH (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00011</td>
<td></td>
</tr>
</tbody>
</table>
Advanced SIMD modified immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

Advanced SIMD shift by immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

The following constraints also apply to this encoding: immh != 0000 && immh != 0000
### Decode fields opcode Instruction Details

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1011x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 00000</td>
<td>SSHR</td>
</tr>
<tr>
<td>0 00010</td>
<td>SSRA</td>
</tr>
<tr>
<td>0 00100</td>
<td>SRSHR</td>
</tr>
<tr>
<td>0 00110</td>
<td>SRSRA</td>
</tr>
<tr>
<td>0 01000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 01010</td>
<td>SHL</td>
</tr>
<tr>
<td>0 01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 01110</td>
<td>SQSHL (immediate)</td>
</tr>
<tr>
<td>0 10000</td>
<td>SHRN, SHRN2</td>
</tr>
<tr>
<td>0 10001</td>
<td>RSHRN, RSHRN2</td>
</tr>
<tr>
<td>0 10010</td>
<td>SQSHRN, SQSHRN2</td>
</tr>
<tr>
<td>0 10011</td>
<td>SORSHRN, SORSHRN2</td>
</tr>
<tr>
<td>0 10100</td>
<td>SSHLL, SSHLL2</td>
</tr>
<tr>
<td>0 11100</td>
<td>SCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td>0 11111</td>
<td>FCVTZS (vector, fixed-point)</td>
</tr>
<tr>
<td>1 00000</td>
<td>USHR</td>
</tr>
<tr>
<td>1 00010</td>
<td>USRA</td>
</tr>
<tr>
<td>1 00100</td>
<td>URSHR</td>
</tr>
<tr>
<td>1 00110</td>
<td>URSRA</td>
</tr>
<tr>
<td>1 01000</td>
<td>SRI</td>
</tr>
<tr>
<td>1 01010</td>
<td>SLI</td>
</tr>
<tr>
<td>1 01100</td>
<td>SOSHLU</td>
</tr>
<tr>
<td>1 01110</td>
<td>UOSHL (immediate)</td>
</tr>
<tr>
<td>1 10000</td>
<td>SQSHRUN, SQSHRUN2</td>
</tr>
<tr>
<td>1 10001</td>
<td>SORSHRUN, SORSHRUN2</td>
</tr>
<tr>
<td>1 10010</td>
<td>UOSHRN, UOSHRN2</td>
</tr>
<tr>
<td>1 10011</td>
<td>UQRSHRN, UQRSHRN2</td>
</tr>
<tr>
<td>1 10100</td>
<td>USHLL, USHLL2</td>
</tr>
<tr>
<td>1 11100</td>
<td>UCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td>1 11111</td>
<td>FCVTZU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

#### Advanced SIMD vector x indexed element

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>U</td>
<td>Decode fields</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>----</td>
<td>---------------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>SMLAL, SMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>SMLSL, SMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>MUL (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>SMULL, SMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>SQDMULL, SQDMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>SQDMULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>SQRDMULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>SDOT (by element)</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>0x</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0x</td>
<td>0100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>FMLA (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>FMLS (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>FMUL (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>SUDOT (by element)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>01</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>BFDOT (by element)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1x</td>
<td>0001</td>
<td>FMLA (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>0101</td>
<td>FMLS (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>FMUL (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0000</td>
<td>FMLAL, FMLAL2 (by element) — FMLAL</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>10</td>
<td>0100</td>
<td>FMLSL, FMLSL2 (by element) — FMLSL</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>10</td>
<td>1111</td>
<td>USDOT (by element)</td>
<td>FEAT_I8MM</td>
</tr>
<tr>
<td>11</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1111</td>
<td>BFMLALB, BFMLALT (by element)</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>MLA (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>UMLAL, UMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>MLS (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>UMLSL, UMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>UMULL, UMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>SQRDMLAH (by element)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>UDOT (by element)</td>
<td>FEAT_DotProd</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>SQRDMLSH (by element)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1</td>
<td>0x1000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x1100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>000011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>000101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>000111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0001001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0001011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>001001</td>
<td>FMULX (by element) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>010xx1</td>
<td>FCMIA (by element)</td>
<td>FEAT_FCMA</td>
</tr>
<tr>
<td>1</td>
<td>1xx1001</td>
<td>FMULX (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10xx1</td>
<td>FCMLA (by element)</td>
<td>FEAT_FCMA</td>
</tr>
</tbody>
</table>
### Cryptographic three-register, imm2

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1000</td>
<td>FMLAL, FMLAL2 (by element) — FMLAL2</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1100</td>
<td>FMLSL, FMLSL2 (by element) — FMLSL2</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Cryptographic three-register SHA 512

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td></td>
<td>SM3TT1A</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td></td>
<td>SM3TT1B</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td></td>
<td>SM3TT2A</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td></td>
<td>SM3TT2B</td>
<td>FEAT SM3</td>
</tr>
</tbody>
</table>

### Cryptographic four-register

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Op0</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td></td>
<td>FOR3</td>
<td>FEAT SHA3</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td></td>
<td>BCAX</td>
<td>FEAT SHA3</td>
</tr>
</tbody>
</table>
**Cryptographic two-register SHA 512**

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

```
<table>
<thead>
<tr>
<th>Op0</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>SM3SS1</td>
<td>FEAT_SM3</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
```

**Conversion between floating-point and fixed-point**

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>scale</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 32-bit to single-precision</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 32-bit to single-precision</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — single-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — single-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 32-bit to double-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 32-bit to double-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>00</td>
<td>0</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — double-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>01</td>
<td>00</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — double-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>01</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
```
### Instruction Details

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>scale</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — half-precision to 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — half-precision to 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 64-bit to single-precision</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 64-bit to single-precision</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — single-precision to 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — single-precision to 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 64-bit to double-precision</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 64-bit to double-precision</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — double-precision to 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — double-precision to 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td></td>
<td>SCVTF (scalar, fixed-point) — 64-bit to half-precision</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td></td>
<td>UCVTF (scalar, fixed-point) — 64-bit to half-precision</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td></td>
<td>FCVTZS (scalar, fixed-point) — half-precision to 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>001</td>
<td></td>
<td>FCVTZU (scalar, fixed-point) — half-precision to 64-bit</td>
</tr>
</tbody>
</table>

### Conversion between floating-point and integer

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

#### Top-level encodings for A64

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>000</td>
<td>FCVTNS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>001</td>
<td>FCVTNU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>SCVTU (scalar, integer) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>UCVTF (scalar, integer) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>FCVTAS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>FCVTAU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>FMOV (general) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>ptype</td>
<td>rmode</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>---------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 111</td>
<td>FMOV (general) — 32-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01 000</td>
<td>FCVTSP (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01 001</td>
<td>FCVTU (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 1x 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 10 000</td>
<td>FCVTMS (scalar) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11 001</td>
<td>FCVTZU (scalar, integer) — single-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0x 11x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 000</td>
<td>FCVTNS (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 001</td>
<td>FCVTNU (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 010</td>
<td>SCVT (scalar, integer) — 32-bit to double-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 011</td>
<td>UNALLOCATED</td>
<td>FEAT_JSCVT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 000</td>
<td>FCVTAS (scalar) — double-precision to 32-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 000</td>
<td>SCVTSP (scalar) — single-precision to 64-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 001</td>
<td>FCVTU (scalar) — single-precision to 64-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 110</td>
<td>FMOV (general) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 000</td>
<td>FCVTNS (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 00 111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 01 000</td>
<td>FCVTU (scalar) — half-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 01 001</td>
<td>SCVT (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 01 010</td>
<td>UCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 10 100</td>
<td>FCVTAS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00 10 101</td>
<td>FCVTU (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 000</td>
<td>FCVTZS (scalar, integer) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 011</td>
<td>FCVTZU (scalar, integer) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 000</td>
<td>FCVTNS (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 001</td>
<td>FCVTU (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 010</td>
<td>SCVT (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 100</td>
<td>FCVTAS (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 101</td>
<td>FCVTU (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Floating-point data-processing (1 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>000</td>
<td>001</td>
<td>FCVTPU (scalar) — single-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>010</td>
<td>000</td>
<td>FCVTMS (scalar) — single-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>010</td>
<td>001</td>
<td>FCVTMU (scalar) — single-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>000</td>
<td>000</td>
<td>FCVTZS (scalar, integer) — single-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>001</td>
<td>001</td>
<td>FCVTZU (scalar, integer) — single-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>x1</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>000</td>
<td>000</td>
<td>FCVTNS (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>000</td>
<td>001</td>
<td>FCVTNU (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>000</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 64-bit to double-precision</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>000</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 64-bit to double-precision</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>000</td>
<td>100</td>
<td>FCVTAS (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>000</td>
<td>101</td>
<td>FCVTAU (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>110</td>
<td>FMOV (general) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>111</td>
<td>FMOV (general) — 64-bit to double-precision</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>000</td>
<td>FCVTPS (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>001</td>
<td>FCVTPU (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1x</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>000</td>
<td>FCVTMS (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>001</td>
<td>FCVTMU (scalar) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, integer) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, integer) — double-precision to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>x0</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>01</td>
<td>110</td>
<td>FMOV (general) — top half of 128-bit to 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>01</td>
<td>111</td>
<td>FMOV (general) — 64-bit to top half of 128-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1x</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
| 1  | 0 | 11             | 00    | 000    | FCVTNS (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 00    | 001    | FCVTNU (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 00    | 010    | SCVTF (scalar, integer) — 64-bit to half-precision | FEAT FP16
| 1  | 0 | 11             | 00    | 011    | UCVTF (scalar, integer) — 64-bit to half-precision | FEAT FP16
| 1  | 0 | 11             | 00    | 100    | FCVTAS (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 00    | 101    | FCVTAU (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 00    | 110    | FMOV (general) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 00    | 111    | FMOV (general) — 64-bit to half-precision | FEAT FP16
| 1  | 0 | 11             | 01    | 000    | FCVTPS (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 01    | 001    | FCVTPU (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 10    | 000    | FCVTMS (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 10    | 001    | FCVTMU (scalar) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 11    | 000    | FCVTZS (scalar, integer) — half-precision to 64-bit | FEAT FP16
| 1  | 0 | 11             | 11    | 001    | FCVTZU (scalar, integer) — half-precision to 64-bit | FEAT FP16
<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000001</td>
<td>FMOV (register) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000010</td>
<td>FABS (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000011</td>
<td>FNEG (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000101</td>
<td>FCVT — single-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001000</td>
<td>FRINTN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001001</td>
<td>FRINTP (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001010</td>
<td>FRINTM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001011</td>
<td>FRINTZ (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001101</td>
<td>FRINTX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001110</td>
<td>FRINTI (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010000</td>
<td>FRINT32Z (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010001</td>
<td>FRINT32X (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010010</td>
<td>FRINT64Z (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010011</td>
<td>FRINT64X (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0101XX</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011XXX</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>FMOV (register) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000001</td>
<td>FABS (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000010</td>
<td>FNEG (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000011</td>
<td>FSQRT (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000101</td>
<td>BFCVT</td>
<td>FEAT_BF16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000111</td>
<td>FCVT — double-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001000</td>
<td>FRINTN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001001</td>
<td>FRINTP (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001010</td>
<td>FRINTM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001011</td>
<td>FRINTZ (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001100</td>
<td>FRINTA (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001110</td>
<td>FRINTX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001111</td>
<td>FRINTI (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010000</td>
<td>FRINT32Z (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010001</td>
<td>FRINT32X (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010010</td>
<td>FRINT64Z (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010011</td>
<td>FRINT64X (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0101XX</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011XXX</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0XXXXX</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>S</td>
<td>ptype</td>
<td>opcode</td>
<td>Instruction Details</td>
<td>Feature</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-------</td>
<td>--------</td>
<td>--------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000000</td>
<td>FMOV (register) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000001</td>
<td>FABS (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000010</td>
<td>FNEG (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000011</td>
<td>FSQRT (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000100</td>
<td>FCVT — half-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000101</td>
<td>FCVT — half-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001000</td>
<td>FRINTN (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001001</td>
<td>FRINTP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001010</td>
<td>FRINTM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001011</td>
<td>FRINTZ (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001100</td>
<td>FRINTA (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001110</td>
<td>FRINTX (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001111</td>
<td>FRINTI (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Floating-point compare

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

---

### Instruction Details

**Top-level encodings for A64**

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>op</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>000000</td>
<td>FCMP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010000</td>
<td>FCMP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>100000</td>
<td>FCMPE</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>000000</td>
<td>FCMP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010000</td>
<td>FCMP</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>100000</td>
<td>FCMPE</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>100000</td>
<td>FCMPE</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>110000</td>
<td>FCMPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>000000</td>
<td>FCMP</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010000</td>
<td>FCMP</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>100000</td>
<td>FCMPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>110000</td>
<td>FCMPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
Floating-point immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>xxxxx1 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>xxx1xx UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>x1xxx UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1xxxx UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 00000</td>
<td>FMOV (scalar, immediate) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 010000</td>
<td>FMOV (scalar, immediate) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11000000</td>
<td>FMOV (scalar, immediate) — half-precision FEAT FP16</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Floating-point conditional compare

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 00</td>
<td>FCCMP — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 01</td>
<td>FCCMPE — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 11</td>
<td>FCCMPE — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11 0</td>
<td>FCCMP — half-precision FEAT FP16</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11 1</td>
<td>FCCMPE — half-precision FEAT FP16</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Floating-point data-processing (2 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1xx1 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1x1x UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>11xx UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>10 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1 UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 00000</td>
<td>FMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
</tbody>
</table>
## Floating-point conditional select

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M 0 0 0 0 0 0 0 0 1</td>
<td>FDIV (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 0 0</td>
<td>FDIV (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 0 1</td>
<td>FDIV (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 1 0 0</td>
<td>FADD (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 1 1 1</td>
<td>FADD (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 0 0 0</td>
<td>FSUB (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 0 1 0</td>
<td>FSUB (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 1 0 0</td>
<td>FMAX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 1 1 0</td>
<td>FMAXNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 1 0 0 0 0</td>
<td>FMIN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 1 0 1 0 0</td>
<td>FMINNM (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 0 0 0</td>
<td>FMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 0 0 1</td>
<td>FMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 0 1 0 0</td>
<td>FADD (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 1 0 0 0</td>
<td>FADD (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 1 1 0 0</td>
<td>FSUB (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 1 1 1 0</td>
<td>FSUB (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 0 0 0 0 0</td>
<td>FMAX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 0 0 0 0 1</td>
<td>FMAXNM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 0 0 1 0 0</td>
<td>FMIN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 0 1 0 0 0</td>
<td>FMINNM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 1 0 0 0 0</td>
<td>FNMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 1 0 0 0 1</td>
<td>FNMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 1 0 0 0 0 0 0 0</td>
<td>FMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 1 0 0 0 0 0 0 1</td>
<td>FMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 0 0 0 0</td>
<td>FADD (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 0 0 0 1</td>
<td>FADD (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 1 0 0 0</td>
<td>FSUB (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 1 0 0 1</td>
<td>FSUB (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 0 0 0 0</td>
<td>FMAX (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1 0 0 0 0 0 1</td>
<td>FMAXNM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 0 0 0 0 0</td>
<td>FMIN (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 0 0 0 0 1</td>
<td>FMINNM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 1 0 0 0 0</td>
<td>FNMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1 1 1 0 0 0 1</td>
<td>FNMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>

## Floating-point data-processing (3 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M 0 0 0 0 0 0 0 0 1 0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 1 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 0 1 0</td>
<td>FCSEL — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 0 1 0 1</td>
<td>FCSEL — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M 0 0 0 0 0 0 1 1 0</td>
<td>FCSEL — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>M 0 0 0 0 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>S</td>
<td>ptype</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>--------</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages
Pseudocodes
// AArch32.AT()
// ============
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, ATAccess ataccess)
TranslationStage stage = stage_in;
SecurityState ss;
Regime regime;
boolean eae;

// ATS1Hx instructions
if el == EL2 then
    regime = Regime_EL2;
eae = TRUE;
ss = SS_NonSecure;

// ATS1Cxx instructions
elsif stage == TranslationStage_1 || (stage == TranslationStage_12 && !HaveEL(EL2)) then
    stage = TranslationStage_1;
    ss = SecurityStateAtEL(PSTATE.EL);
    regime = if ss == SS_Secure && ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL1;
eae = TTBCR.EAE == '1';

// ATS12NSOxx instructions
else
    regime = Regime_EL1;
eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR.EAE == '1';
ss = SS_NonSecure;

AddressDescriptor addrdesc;
aligned = TRUE;
ispriv = el != EL0;
supersection = '0';
iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;

// Prepare fault fields in case a fault is detected
fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

if eae then
    (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, vaddress, acctype, aligned, iswrite, ispriv);
else
    (fault, addrdesc, sdftype) = AArch32.S1TranslateSD(fault, regime, ss, vaddress, acctype, aligned, iswrite, ispriv);
    supersection = if sdftype == SDFType_Supersection then '1' else '0';

// ATS12NSOxx instructions
if stage == TranslationStage_12 && fault.statuscode == Fault_None then
    s2fs1walk = FALSE;
    (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
    // Take exception when External abort occurs on translation table walk
    if (IsExternalAbort(fault) || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL1 && EL2Enabled() && fault.s2fs1walk)) then
        PAR = bits(64) UNKNOWN;
        AArch32.Abort(vaddress, fault);

addrdesc.fault = fault;

if (eae || (stage == TranslationStage_12 && (HCR.VM == '1' || HCR.DC == '1')) || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL2)) then
    AArch32.EncodePARLD(addrdesc, ss);
else
    AArch32.EncodePARSD(addrdesc, supersection, ss);
return;
Library pseudocode for aarch32/at/AArch32.EncodePARLD

// AArch32.EncodePARLD()
// =====================
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

    if !IsFault(addrdesc) then
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        PAR.F = '0';
        PAR.SH = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR.NS = ns;
        PAR.PA = addrdesc.paddress.address<39:12>;
        PAR.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
    else
        PAR.F = '1';
        PAR.FST = AArch32.PARFaultStatusLD(addrdesc.fault);
        PAR.S2WLK = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR.FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
        PAR.LPAE = '1';
        PAR<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";  // IMPDEF
    return;

Library pseudocode for aarch32/at/AArch32.EncodePARSD

// AArch32.EncodePARSD()
// =====================
// Returns 32-bit format PAR on address translation instruction.

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, SecurityState ss)

    AddressDescriptor addrdesc = addrdesc_in;
    if !IsFault(addrdesc) then
        if (addrdesc.memattrs.memtype == MemType_Device || (addrdesc.memattrs.inner.attrs == MemAttr_NC && addrdesc.memattrs.outer.attrs == MemAttr_NC)) then
            addrdesc.memattrs.shareability = Shareability_OSH;
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        bits(2) sh = if addrdesc.memattrs.shareability != Shareability_NSH then '01' else '00';
        PAR.F = '0';
        PAR.SS = supersection;
        PAR.Outer = AArch32.ReportedOuterAttrs(AArch32.PAROuterAttrs(addrdesc.memattrs));
        PAR.Inner = AArch32.ReportedInnerAttrs(AArch32.PARInnerAttrs(addrdesc.memattrs));
        PAR.SH = ReportedPARShareability(sh);
        PAR<8> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";  // IMPDEF
        PAR.NS = ns;
        PAR.NOS = if addrdesc.memattrs.shareability == Shareability_OSH then '0' else '1';
        PAR.LPAE = '0';
        PAR.PA = addrdesc.paddress.address<39:12>;
    else
        PAR.F = '1';
        PAR.FST = AArch32.PARFaultStatusSD(addrdesc.fault);
        PAR.LPAE = '0';
        PAR<31:16> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";  // IMPDEF
    return;
Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

// AArch32.PARFaultStatusLD()
// =========================
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(32) syndrome;
    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome<1:0> = if fault.level == 1 then '01' else '10';
        syndrome<5:2> = '1111';
    else
        syndrome = AArch32.FaultStatusLD(TRUE, fault);
    return syndrome<5:0>;

Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

// AArch32.PARFaultStatusSD()
// ==========================
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
    bits(32) syndrome;
    syndrome = AArch32.FaultStatusSD(TRUE, fault);
    return syndrome<12,10,3:0>;

Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

// AArch32.PARInnerAttrs()
// ========================
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;
    if memattrs.membtype == MemType_Device then
        if memattrs.device == DeviceType_nGnRnE then
            result = '001'; // Non-cacheable
        elsif memattrs.device == DeviceType_nGnRE then
            result = '011'; // Non-cacheable
        else
            MemAttrHints inner = memattrs.inner;
            if inner.attrs == MemAttr_NC then
                result = '000'; // Non-cacheable
            elseif inner.attrs == MemAttr_WB && inner.hints<0> == '1' then
                result = '101'; // Write-Back, Write-Allocate
            elseif inner.attrs == MemAttr_WT then
                result = '110'; // Write-Through
            elseif inner.attrs == MemAttr_WB && inner.hints<0> == '0' then
                result = '111'; // Write-Back, no Write-Allocate
            return result;
        end if
    else
        end if
    end if
Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

// AArch32.PAROuterAttrs()
// =======================
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;
    if memattrs.memtype == MemType_Device then
        result = bits(2) UNKNOWN;
    else
        MemAttrHints outer = memattrs.outer;
        if outer.attrs == MemAttr_NC then
            result = '00'; // Non-cacheable
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elsif outer.attrs == MemAttr_WT && outer.hints<0> == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elsif outer.attrs == MemAttr_WR && outer.hints<0> == '0' then
            result = '11'; // Write-Back, no Write-Allocate
        return result;
AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)

AccType acctype = AccType_DC;
CacheRecord cache;

(cache.acctype = acctype;
cache.cacheop = cacheop;
cache.opscope = opscope;
cache.cachetype = CacheType_Data;
cache.security = SecurityStateAtEL(PSTATE.EL);

if opscope == CacheOpScope_SetWay then
   cache.shareability = Shareability_NSH;
   (cache.set, cache.way, cache.level) = DecodeSW(ZeroExtend(regval), CacheType_Data);

   if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
      ((!ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1') || (ELUsingAArch32(EL2) && HCR.SWIO == '1') ||
       (ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))) then
      cache.cacheop = CacheOp_CleanInvalidate;
   
   CACHE_OP(cache);
   return;

if EL2Enabled() then
   if PSTATE.EL IN {EL0, EL1} then
      cache.is_vmid_valid = TRUE;
      cache.vmid = VMID[];
   else
      cache.is_vmid_valid = FALSE;
   else
      cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
   cache.is_asid_valid = TRUE;
   cache.asid = ASID[];
else
   cache.is_asid_valid = FALSE;

need_translate = DCInstNeedsTranslation(opscope);
iswrite = cacheop == CacheOp_Invalidate;
vaddress = regval;

size = 0;       // by default no watchpoint address
if iswrite then
   size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
   assert size >= 4*(2^UInt(CTR_EL0.DminLine))) && size <= 2048;
   assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
vaddress = Align(regval, size);

cache.translated = need_translate;

if need_translate then
   memaddrdesc = AArch32.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
   if IsFault(memaddrdesc) then
      AArch32.Abort(regval, memaddrdesc.fault);
   memattrs = memaddrdesc.memattrs;
   cache.paddress = memaddrdesc.paddress;
   if opscope == CacheOpScope_PoC then
      cache.shareability = memattrs.shareability;
   else
      cache.shareability = Shareability_NSH;
else
   cache.shareability = Shareability_UNKNOWN;
   cache.paddress = FullAddress UNKNOWN;
if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled()
   && (!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') ||
   (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00')) then
   CacheOp_CleanInvalidate;

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)
boolean match;
if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
   // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
   match_word = Zeros(32);
   if vaddress<31:5> == ExcVectorBase()<31:5> then
      if HaveEL(EL3) && !IsSecure() then
         match_word<UInt(vaddress<4:2>) + 24> = '1';  // Non-secure vectors
      else
         match_word<UInt(vaddress<4:2>) + 0> = '1';  // Secure vectors (or no EL3)
   else
      if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
         match_word<UInt(vaddress<4:2>) + 8> = '1';       // Monitor vectors
   // Mask out bits not corresponding to vectors.
   bits(32) mask;
   if !HaveEL(EL3) then
      mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
   elsif !ELUsingAArch32(EL3) then
      mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
   else
      mask = '11011110':'00000000':'11011100':'11011110';
   match_word = match_word AND DBGVCR AND mask;
   match = !IsZero(match_word);

   // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
   if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
      match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
   else
      match = FALSE;
   return match;

Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
// the state of the (DBGEN AND SPIDEN) signal.
if !HaveEL(EL3) && !IsSecure() then return FALSE;
return DBGEN == HIGH && SPIDEN == HIGH;
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,
integer size)
assert ELUsingAArch32(S1TranslationRegime());
assert n < NumBreakpointsImplemented();

enabled = DBGBCR[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
linked, DBGBCR[n].LBN, isbreakpnt, ispriv);
(value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

if size == 4 then // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match the address of the
    // second halfword of an instruction, but not the address of the first halfword, it is
    // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
    // event.
    (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
    if !value_match && match_i then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if value_mismatch && !mismatch_i then
        value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
    // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
    // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
    // at the address DBGVR[n]+2.
    if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

match = value_match && state_match && enabled;
mismatch = value_mismatch && state_match && enabled;
return (match, mismatch);
Library pseudocode for aarch32/debug/breakpoint-AArch32.BreakpointValueMatch
// AArch32.BreakpointValueMatch()
// -----------------------------
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

   // "n" is the identity of the breakpoint unit to match against.
   // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
   // matching breakpoints.
   // "linked_to" is TRUE if this is a call from StateMatch for linking.
   integer n = n_in;

   // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
   // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
   if n >= NumBreakpointsImplemented() then
      Constraint c;
      (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMPL);
      assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
      if c == Constraint_DISABLED then return (FALSE,FALSE);

   // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
   // call from StateMatch for linking).
   if DBGBCR[n].E == '0' then return (FALSE,FALSE);

   context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

   // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
   dbgtype = DBGBCR[n].BT;
   if ((dbgtype IN {'011x','11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) ||    // Context matching
      (dbgtype == '010x' && HaltOnBreakpointOrWatchpoint()) ||                  // Address mismatch
      (dbgtype != '0x0x' && !context_aware) ||                                  // Context matching
      (dbgtype == '1xxx' && !HaveEL(EL2))) then                                 // EL2 extension
      (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
      assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
      if c == Constraint_DISABLED then return (FALSE,FALSE);
   // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

   // Determine what to compare against.
   match_addr = (dbgtype == '0x0x');
   mismatch = (dbgtype == '010x');
   match_vmid = (dbgtype == '10xx');
   match_cid1 = (dbgtype == 'xx1x');
   match_cid2 = (dbgtype == '11xx');
   linked = (dbgtype == 'xxx1');

   // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
   // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
   // code can just test for match_addr == TRUE to confirm all these things.
   if linked_to && (!linked || !match_addr) then return (FALSE,FALSE);

   // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
   if !linked_to && linked && !match_addr then return (FALSE,FALSE);

   // Do the comparison.
   boolean BVR_match;
   if match_addr then
      boolean byte_select_match;
      byte = UInt(vaddress<1:0>);
      assert byte IN {0,2};                     // "vaddress" is halfword aligned
      byte_select_match = (DBGBCR[n].BAS<byte> == '1');
      integer top = 31;
      BVR_match = (vaddress<top:2> == DBGVR[n]<top:2>) && byte_select_match;
   elsif match_cid1 then
      BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGVR[n]<31:0>);
   elsif match_vmid then
      // "vaddress" is halfword aligned
      byte_select_match = (DBGBCR[n].BAS<byte> == '1');
      integer top = 31;
      BVR_match = (vaddress<top:2> == DBGVR[n]<top:2>) && byte_select_match;

Shared Pseudocode Functions
bits(16) vmid;
bits(16) bvr_vmid;
if !ELUsingAArch32(EL2) then
  vmid = ZeroExtend(VTTBR.VMID, 16);
  bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
  vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
  bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
else
  vmid = VTTBR_EL2.VMID;
  bvr_vmid = DBGBXVR[n]<15:0>;
  BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                 vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug()) &&
                !ELUsingAArch32(EL2) &&
                DBGBXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

  bvr_match_valid = (match_addr || match_cid1);
  bxvr_match_valid = (match_vmid || match_cid2);

  match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

  return (match && !mismatch, !match && mismatch);
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC_in, bit HMC_in,
                          bits(2) PxC_in, boolean linked_in, bits(4) LBN,
                          boolean isbreakpnt, boolean ispriv)

  // "SSC_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
  // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
  // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
  // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
  // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
  bits(2) SSC = SSC_in;
  bit HMC = HMC_in;
  bits(2) PxC = PxC_in;
  boolean linked = linked_in;

  // If parameters are set to a reserved type, behaves as either disabled or a defined type
  Constraint c;
  (c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
  if c == ConstraintDISABLED then return FALSE;

  // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
  // CheckValidStateMatch are valid.

  PL2_match = HaveEL(EL2) & (HMC == '1' & (SSC:PxC != '1000') || SSC == '11');
  PL1_match = PxC<0> == '1';
  PL0_match = PxC<1> == '1';
  SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

  boolean priv_match;
  if !ispriv && !isbreakpnt then
    priv_match = PL0_match;
  elsif SSU_match then
    priv_match = PSTATE.M IN {M32_User, M32_Svc, M32_System};
  else
    case PSTATE.EL of
      when EL3 priv_match = PL1_match;           // EL3 and EL1 are both PL1
      when EL2 priv_match = PL2_match;
      when EL1 priv_match = PL1_match;
      when EL0 priv_match = PL0_match;
    end

  boolean security_state_match;
  ss = CurrentSecurityState();
  case SSC of
    when '00' security_state_match = TRUE;                             // Both
    when '01' security_state_match = ss == SS_NonSecure;               // Non-secure only
    when '10' security_state_match = ss == SS_Secure;                 // Secure only
    when '11' security_state_match = (HMC == '1' || ss == SS_Secure); // HMC=1 -> Both, 0 -> Secure
  end

  integer lbn;
  if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
    last_ctx_cmp = NumBreakpointsImplemented() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
      (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
      assert c IN {ConstraintDISABLED, ConstraintNONE, ConstraintUNKNOWN};
      case c of
        when ConstraintDISABLED return FALSE; // Disabled
        when ConstraintNONE linked = FALSE;    // No linking
      end
    else
      // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

  boolean linked_match;
  if linked then
    vaddress = bits(32) UNKNOWN;
    linked_to = TRUE;
    (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);
return priv_match && security_state_match && (!linked || linked_match);

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

// AArch32.GenerateDebugExceptions()
// =================================
boolean AArch32.GenerateDebugExceptions()
return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

// AArch32.GenerateDebugExceptionsFrom()
// =====================================
boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)
// EL0 - EL3, EL1 = EL3, EL2 = EL3, EL3 = EL3
boolean enabled;
if !ELUsingAArch32(DebugTargetFrom(secure)) then
    mask = '0'; // No PSTATE.D in AArch32 state
    return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);
if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
    return FALSE;

boolean enabled;
if HaveEL(EL3) && secure then
    assert from != EL2; // Secure EL2 always uses AArch64
    if IsSecureEL2Enabled() then
        // Implies that EL3 and EL2 both using AArch64
        enabled = MDCR_EL3.SDD == '0';
    else
        spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;

return enabled;
Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch32.CheckForPMUOverflow()
   if !ELUsingAArch32(EL1) then
      AArch64.CheckForPMUOverflow();
      return;
   bit hpme;
   if HaveEL(EL2) then
      hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
   boolean pmuirq;
   bit E;
   pmuirq = PMCR.E == '1' && PMINTENSET.C == '1' && PMOVSSET.C == '1';
   integer counters = GetNumEventCounters();
   if counters != 0 then
      for idx = 0 to counters - 1
         E = if AArch32.PMUCounterIsHyp(idx) then hpme else PMCR.E;
         if E == '1' && PMINTENSET<idx> == '1' && PMOVSSET<idx> == '1' then pmuirq = TRUE;
      SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
      CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
   // The request remains set until the condition is cleared. (For example, an interrupt handler
   // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR.)
```

Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```
// AArch32.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch32.ClearEventCounters()
   if HaveAArch64() then
      // Force the counter to be cleared as a 64-bit counter.
      AArch64.ClearEventCounters();
      return;
   integer counters = AArch32.GetNumEventCountersAccessible();
   if counters != 0 then
      for idx = 0 to counters - 1
         PMEVCNTR[idx] = Zeros();
```
Library pseudocode for aarch32/debug/pmu/AArch32.CountPMUEvents
boolean AArch32.CountPMUEvents(integer idx)
assert idx == CYCLE_COUNTER_ID || idx < GetNumEventCounters();
if !ELUsingAArch32(EL1) then return AArch64.CountPMUEvents(idx);
boolean debug;
boolean enabled;
boolean prohibited;
boolean filtered;
boolean frozen;
boolean resvd_for_el2;
bit E;
bit spme;
bits(32) ovflws;
// Event counting is disabled in Debug state
debug = Halted();
// Software can reserve some counters for EL2
resvd_for_el2 = AArch32.PMUCounterIsHyp(idx);
// Main enable controls
if idx == CYCLE_COUNTER_ID then
    enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
else
    if resvd_for_el2 then
        E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
    else
        E = PMCR.E;
    enabled = E == '1' && PMCNTENSET<idx> == '1';
// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;
// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * One of the following is true:
//   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
//     - FEAT_PMUv3p7 is not implemented
//     - MDCR_EL3.MPMX == 0
//   - EL3 is using AArch32 and SDCR.SPME == 0
// * Not executing at EL0, or SDER.SUNIDEN == 0
if HaveEL(EL3) && IsSecure() then
    spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
    if !ELUsingAArch32(EL3) && HavePMUv3p7() then
        prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
    else
        prohibited = spme == '0';
    if prohibited && PSTATE.EL == EL0 then
        prohibited = SDER.SUNIDEN == '0';
// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * HDCR.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = HDCR.HPMD == '1';
// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();
// Event counting might be frozen
frozen = FALSE;
// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() then
    if HaveEL(EL2) then
        hpmn = if ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
ovflws = ZeroExtend(PMOVSET\(GetNumEventCounters\() - 1:0\));
if resvd_for_el2 then
    FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
    ovflws<UInt(hpmn)-1:0> = Zeros();
else
    FZ = PMCR.FZO;
    if HaveEL(EL2) && \(\text{UInt}(hpmn) < \text{GetNumEventCounters}\()) then
        ovflws<GetNumEventCounters() -1:UInt(hpmn)> = Zeros();
    frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && \(\text{idx} = \text{CYCLE_COUNTER_ID}\) then
    enabled = enabled && (PMCR.DP == '0');
// Otherwise whether event counting is prohibited does not affect the cycle counter
    prohibited = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR.DP.
if HavePMUv3p5() && \(\text{idx} = \text{CYCLE_COUNTER_ID}\) then
    if HaveEL(EL3) && \(\text{IsSecure}\()) then
        sccd = if ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
        if sccd == '1' then prohibited = TRUE;
    if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the \{P, U, NSK, NSU, NSH\} bits
filter = if \(\text{idx} = \text{CYCLE_COUNTER_ID}\) then PMCCFILTR else PMEVTYPER[idx];

P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
    when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';
return !debug && enabled && !prohibited && !filtered && !frozen;

Library pseudocode for aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

// GetNumEventCountersAccessible()
// -------------------------------------
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch32.GetNumEventCountersAccessible()
integer n;
integer total_counters = GetNumEventCounters();
// Software can reserve some counters for EL2
if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
    n = UInt(if ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
    if n > total_counters || (!HaveFeatureHPMN() && n == 0) then
        (-, n) = ConstrainUnpredictableInteger(0, total_counters, Unpredictable_PMUEVENTCOUNTER);
    else
        n = total_counters;
return n;
Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

// AArch32.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
    if HaveAAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        AArch64.IncrementEventCounter(idx, increment);
        return;
    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR[idx]);
    new_value = old_value + PMUCountValue(idx, increment);
    PMEVCNTR[idx] = new_value<31:0>;
    ovflw = 32;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        PMOVSR<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

Library pseudocode for aarch32/debug/pmu/AArch32.PMUCounterIsHyp

// AArch32.PMUCounterIsHyp
// =======================
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch32.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
        hpmn = if !ELUsingAAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        resvd_for_el2 = n >= UInt(hpmn);
        if UInt(hpmn) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(hpmn)) then
            resvd_for_el2 = boolean UNKNOWN;
        else
            resvd_for_el2 = FALSE;
    return resvd_for_el2;
// AArch32.PMUCycle()
// ================================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
if !HavePMUv3() then
    return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if AArch32.CountPMUEvents(idx) then
                accumulated = PMUEventAccumulator[idx];
                AArch32.IncrementEventCounter(idx, accumulated);
                PMUEventAccumulator[idx] = 0;

    integer old_value;
    integer new_value;
    integer ovflw;
    if (AArch32.CountPMUEvents(CYCLE_COUNTER_ID) &&
        (PMCR.LC == '1' || PMCR.D == '0' || HasElapsed64Cycles())) then
        old_value = UInt(PMCCNTR);
        new_value = old_value + 1;
        PMCCNTR = new_value<63:0>;
        ovflw = if PMCR.LC == '1' then 64 else 32;
        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET.C = '1';
            PMOVSR.C = '1';

AArch32.CheckForPMUOverflow();

// AArch32.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
integer counters = AArch32.GetNumEventCountersAccessible();
if counters != 0 then
    for idx = 0 to counters - 1
        if sw_incr<idx> == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
    AArch32_ReportHypEntry(exception);
    AArch32_WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                      // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTRLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSR.ERR = '1';
    UpdateEDSCRFields();

EndOfInstruction();

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32_WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                      // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSR.ERR = '1';
    UpdateEDSCRFields();

EndOfInstruction();

Shared Pseudocode Functions
// AArch32.EnterMonitorModeInDebugState()
// =======================================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()

    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        DLR = bits(32) UNKNOWN;
        DSPSR = bits(32) UNKNOWN;
        EDSCR.ERR = '1';
        UpdateEDSCRFields(); // Update EDSCR processor state flags.
    EndOfInstruction();
// AArch32.WatchpointByteMatch()
// ----------------------------------

doolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    integer top = 31;
    bottom = if DBGWVR[n]<2> == '1' then 2 else 3; // Word or doubleword
    byte select match = (DBGWCR[n].BAS<UInt>(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '1111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte select match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte select_match = ConstrainUnpredictableBool(Unpredictable_WPBACONTIGUOUS);
        end
        bottom = 3; // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE; // Disabled
            when Constraint_NONE mask = 0; // No masking
        end
        // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR match;
    if mask > bottom then
        // If the DBGxVR<n> EL1.RESS field bits are not a sign extension of the MSB
        // of DBGxVR<n> EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        if !IsOnes(DBGBVVR_EL1[n]<63:top>) && !IsZero(DBGBVVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
            end
            WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMAESDBITS);
        end
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;
    end

    return WVR_match && byte select match;
Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());
assert n < NumWatchpointsImplemented();

// "ispriv" is:
// * FALSE for all loads, stores, and atomic operations executed at EL0.
// * FALSE if the access is unprivileged.
// * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR[n].E == '1';
linked = DBGWCR[n].WT == '1';
isbreakpnt = FALSE;

state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

ls_match = FALSE;
ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

value_match = FALSE;
for byte = 0 to size - 1
value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

return value_match && state_match && ls_match && enabled;

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
(HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
(IsDebugException(fault) && MDCR_EL2.TDE == '1'));
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);
if route_to_aarch64 then
AArch64.Abort(ZeroExtend(vaddress), fault);
elif fault.acctype == AccType_IFETCH then
AArch32.TakePrefetchAbortException(vaddress, fault);
else
AArch32.TakeDataAbortException(vaddress, fault);
Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)
    exception = ExceptionSyndrome(exceptype);
    d_side = exceptype == Exception_DataAbort;
    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;
    return exception;

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

// AArch32.CheckPCAlignment()
// =========================

AArch32.CheckPCAlignment()
    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

    // Generate an Alignment fault Prefetch Abort exception
    vaddress = pc;
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    secondstage = FALSE;
    AArch32.Abort(vaddress, AlignmentFault(acctype, iswrite, secondstage));
**Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort**

```java
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = ((TTBCR_S.EAE == '1') ||
                        (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                        (fault.secondstage && boolean IMPLEMENTATION_DEFINED "Stage 2 synchronous external abort reports using Long-descriptor format when TTBCR_S.EAE is 0b0"));
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    bits(32) syndrome;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        bits(32) i_syndrome;
        if (!long_format && boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

        if route_to_monitor then
            DFSR_S = syndrome;
            DFAR_S = vaddress;
        else
            DFSR = syndrome;
            DFAR = vaddress;
    return;
```
Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```c
AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';
    d_side = FALSE;
    bits(32) fsr;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);
    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;
    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```c
AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1}) &&
        (HCR.TGE == '1' ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault));
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;
    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Shared Pseudocode Functions
Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

// AArch32.TakePrefetchAbortException()
// -----------------------------------
AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} && (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) || (IsDebugException(fault) && HDCR.TDE == '1') || IsSecondStage(fault));

    ExceptionRecord exception;
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0C;
    lr_offset = 4;
    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAlignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TranslateException(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    end

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

// AArch32.TakePhysicalFIQException()
// -----------------------------------
AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';
    if route_to_aarch64 then
        AArch64.TranslateException(route_to_monitor, fault, vaddress);
        AArch64.EnterMode(M32_Arithmetic, preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
    else
        AArch32.EnterMode(M32_Arithmetic, preferred_exception_return, lr_offset, vect_offset);
// Take an enabled physical IRQ exception.
AArch32.TakePhysicalIRQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled(EL2) then
  route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
if !route_to_aarch64 && HaveEL(EL2) && !ELUsingAArch32(EL3) then
  route_to_aarch64 = SCR_EL3.IRQ == '1';
if route_to_aarch64 then AArch64.TakePhysicalIRQException();

route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled(EL2) &&
  (HCR.TGE == '1' || HCR.IMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 4;
if route_to_monitor then
  AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
  exception = ExceptionSyndrome(Exception_IRQ);
  AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
  AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
AArch32.TakePhysicalSErrorException()
// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

if !route_to_aarch64 && !EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route to aarch64 then
    AArch64.TakePhysicalSErrorException(full_syndrome);

route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
    (HCR.TGE == '1' || HCR.AMO == '1'));

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;

bits(2) target_el;
if route to monitor then
    target_el = EL3;
elsif PSTATE.EL == EL2 || route to hyp then
    target_el = EL2;
else
    target_el = EL1;

if IsSErrorEdgeTriggered(target_el, full_syndrome) then
    ClearPendingPhysicalSError();

fault = AsyncExternalAbort(parity, pe_error_state, extflag);
vaddress = bits(32) UNKNOWN;

case target_el of
    when EL3
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);  
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    when EL2
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    when EL1
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    otherwise
        Unreachable();
Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```c
// AArch32.TakeVirtualFIQException()
// --------------------------------
AArch32.TakeVirtualFIQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if !ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
    assert HCR.TGE == '0' && HCR.FMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

bits(32) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x1C;
lr_offset = 4;
AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vec_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```c
// AArch32.TakeVirtualIRQException()
// --------------------------------
AArch32.TakeVirtualIRQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if !ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IMO==1
    assert HCR.TGE == '0' && HCR.IMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

bits(32) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x18;
lr_offset = 4;
AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vec_offset);
```
Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualSErrorException

// AArch32.TakeVirtualSErrorException()
// ----------------------------------------

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) pe_error_state, bits(25) full_syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAAArch32(EL2) then // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAAArch32(EL1) then AArch64.TakeVirtualSErrorException(full_syndrome);

    route_to_monitor = FALSE;
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;
    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    FaultRecord fault;
    if HaveRASExt() then
        if ELUsingAAArch32(EL2) then
            fault = AsyncExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AsyncExternalAbort(FALSE, VSESR_EL2.AET, VSESR_EL2.ExT);
        else
            fault = AsyncExternalAbort(parity, pe_error_state, extflag);

    ClearPendingVirtualSErrorException();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

// AArch32.SoftwareBreakpoint()
// -----------------------------

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && ELUsingAAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;        // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;
    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);

Library pseudocode for aarch32/exceptions/debug/DebugException

constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';
Library pseudocode for aarch32/exceptions/exceptions/AArch32.CheckAdvSIMDOrFPRegisterTraps

// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;
        if (tid0 == '1' && reg == '0000')                             // FPSID
            || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then    // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception_Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap                ec = 0x01;
        when Exception_CP15RRTTrap            ec = 0x03;
        when Exception_CP15RRTTrap            ec = 0x04;
        when Exception_CP14RTTrap             ec = 0x05;
        when Exception_CP14RRTTrap             ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap    ec = 0x07;
        when Exception_FPIDTrap               ec = 0x08;
        when Exception_PACTrap               ec = 0x09;
        when Exception_CP14DTTrap             ec = 0x0C;
        when Exception_IllegalState           ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        ec = 0x11;
        when Exception_HypervisorCall        ec = 0x12;
        when Exception_MonitorCall           ec = 0x13;
        when Exception/InstructionAbort       ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAignment            ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort             ec = 0x24;
        when Exception_NV2DataAbort          ec = 0x25;
        when Exception_FPTrappedException     ec = 0x28;
        otherwise Unreachable();

        if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
            ec = ec + 1;
        bit il;
        if il_is_valid then
            il = if ThisInstrLength() == 32 then '1' else '0';
        else
            il = '1';

        return (ec,il);
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
{
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
             (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
}

// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)
{
    Exception exceptype = exception.exceptype;
    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
}

// AArch32.ResetControlRegisters(boolean cold_reset);
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
// AArch32.TakeReset()
// ===============
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0'; // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    else
        PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');
    boolean branch_condition = FALSE;
    BranchTo(rv, BranchType_RESET, branch_condition);

// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```c
// AArch32.FPTrappedException()
// ================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX = '1';
    FPEXC.TFV = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;  // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8> = '111';  // VECITR is RES1
    AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```c
// AArch32.CallHypervisor()
// ================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);
    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```c
// AArch32.CallSupervisor()
// ================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

```c
// AArch32.TakeHVCException()
// ================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```
Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

// AArch32.TakeSMCException()
// =========================

AArch32.TakeSMCException()
assert HaveEL(EL3) & ELUsingAArch32(EL3); AArch32.ITAdvance(); SSAdvance();
bits(32) preferred_exception_return = NextInstrAddr(); vect_offset = 0x08;
lr_offset = 0;
AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

// AArch32.TakeSVCException()
// =========================

AArch32.TakeSVCException(bits(16) immediate)
AArch32.ITAdvance(); SSAdvance();
rout_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
bits(32) preferred_exception_return = NextInstrAddr(); vect_offset = 0x08;
lr_offset = 0;
if PSTATE.EL == EL2 || rout_to_hyp then
  exception = ExceptionSyndrome(Exception_SupervisorCall);
  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
  AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return, integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTL.R.TE;                             // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR.Gen[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR.Gen[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR.Gen[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSBSExt() then PSTATE.SSBS = HSCTL.R.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);
    CheckExceptionCatch(TRUE);                    // Check for debug event on exception entry
    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCCTL.R.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    endif
    PSTATE.E = SCCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCCTL.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSBSExt() then PSTATE.SSBS = SCCTL.R.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);
    CheckExceptionCatch(TRUE);                    // Check for debug event on exception entry
    EndOfInstruction();
```
AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)

    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                           // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
        branch_conditional = FALSE;
        BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);
    CheckExceptionCatch(TRUE);                     // Check for debug event on exception entry
    EndOfInstruction();
AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)

if PSTATE.EL == EL0 && (!EL2Enabled() || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) && !ELUsingAArch32(EL1) then
    // The PE behaves as if FPEXC.EN is 1
    AArch64.CheckFPEnabled();
    AArch64.CheckFPAdvSIMDEnabled();
else if PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL1) then
    if fpexc_check && HCR_EL2.RW == '0' then
        fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
        if fpexc_en == '0' then UNDEFINED;
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;
        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';
        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;
            // Check if access disabled in CPACR
            boolean disabled;
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                when '11' disabled = FALSE;
                if disabled then UNDEFINED;
            if required, check FPEXC enabled bit.
            if fpexc_check && FPEXC.EN == '0' then UNDEFINED;
        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPT and CPTR_EL3
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```java
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.
AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;
            if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';
            // Check if access disabled in HCPT
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                exception.syndrome<24:20> = ConditionSyndrome();
                if advsimd then
                    exception.syndrome<5> = '1';
                else
                    exception.syndrome<5> = '0';
                    exception.syndrome<3:0> = '1010';       // coproc field, always 0xA
                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(exception);
                else
                    AArch32.TakeHypTrapException(exception);
            return;
        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then
                AArch64.AdvSIMDFPAccessTrap(EL3);
        return;
```
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
if HaveFGTExt() then
    route_to_el2 = FALSE;
if PSTATE.EL == EL0 then
    route_to_el2 = (ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                   (HCR_EL2.<E2H, TGE> != '1' && (!HaveEL(EL1) || SCR_EL3.FGTEn == '1')));
if route_to_el2 then
    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;
    exception.trappedsyscallinst = TRUE;
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFxTrap

// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
assert HaveEL(target_el);

// Check for routing to AArch64
if !ELUsingAArch32(target_el) then
    AArch64.CheckForWFxTrap(target_el, wfxtype);
return;

boolean is_wfe = wfxtype == WFxType_WFE;
boolean trap;
case target_el of
    when EL1
        trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
    when EL2
        trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
    when EL3
        trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';
if trap then
    if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
        AArch64.WFxTrap(wfxtype, target_el);
    if target_el == EL3 then
        AArch32.TakeMonitorTrapException();
    elsif target_el == EL2 then
        exception = ExceptionSyndrome(Exception_WFxTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
    case wfxtype of
        when WFxType_WFI
            exception.syndrome<0> = '0';
        when WFxType_WFE
            exception.syndrome<0> = '1';
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;
        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle([NextInstrAddr()], 2, AccType_IFETCH, TRUE);
        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxx', '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
taken on the IT instruction or the next instruction. This is not reflected in
// the pseudocode, which always takes the exception on the IT instruction. This
// also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;
        return;

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;
        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
    if setend_disabled == '1' then
        UNDEFINED;
    return;
// AArch32.SystemAccessTrap()
// =========================
// Trapped system register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
(valid, target_el) = ELFromM32(mode);
assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

if target_el == EL2 then
  exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
  AArch32.TakeHypTrapException(exception);
else
  AArch32.TakeUndefInstrException();
Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

// AArch32.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
ExceptionRecord exception;

case ec of
when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
otherwise    unreachable();

bits(20) iss = Zerps();

if exception.exceptype == Exception_Uncategorized then
  return exception;
elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
  // Trapped MRC/MCR, VMRS on FPSID
  iss<13:10> = instr<19:16>;     // CRn, Reg in case of VMRS
  iss<8:5> = instr<15:12>;      // Rt
  iss<9> = '0';                  // RES0
  if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
    iss<19:17> = instr<7:5>;     // opc2
    iss<16:14> = instr<23:21>;   // opc1
    iss<4:1> = instr<3:0>;       // CRm
  else //VMRS Access
    iss<19:17> = '000';          //opc2 - Hardcoded for VMRS
    iss<16:14> = '111';          //opc1 - Hardcoded for VMRS
    iss<4:1> = '0000';           //CRm  - Hardcoded for VMRS
  elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
    // Trapped MRRC/MCRR, VMRS/VMSR
    iss<19:16> = instr<7:4>;     // opc1
    iss<13:10> = instr<19:16>;   // Rt2
    iss<8:5> = instr<15:12>;     // Rt
    iss<4:1> = instr<3:0>;       // CRm
  elsif exception.exceptype == Exception_CP14DTTrap then
    // Trapped LDC/STC
    iss<19:12> = instr<7:0>;     // immB
    iss<4:0> = instr<23>;         // U
    iss<2:1> = instr<24,21>;     // P,W
    if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
      iss<8:5> = bits(4) UNKNOWN;
      iss<3> = '1';               // Direction
    iss<0> = instr<20>;           // Direction
  exception.syndrome<24:20> = Conditionsyndrome();
  exception.syndrome<19:0> = iss;
  return exception;
Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) &amp; !IsSecure() &amp; ELUsingAArch32(EL2);
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) &amp; ELUsingAArch32(EL3);
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)
    route_to_hyp = PSTATE.EL == EL0 &amp; EL2Enabled() &amp; HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```plaintext
// AArch32.UndefineFault()
// ========================
AArch32.UndefineFault()
```

```plaintext
if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefineFault();
AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```plaintext
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
```

```plaintext
assert statuscode != Fault_None;
```

```plaintext
case statuscode of
    when Fault_Domain return TRUE;
    when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
        return level == 2;
    otherwise return FALSE;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

```plaintext
// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
```

```plaintext
assert fault.statuscode != Fault_None;
```

```plaintext
bits(32) fsr = Zeros();
```

```plaintext
if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
if d_side then
    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
        fsr<13> = '1'; fsr<11> = '1';
    else
        fsr<11> = if fault.write then '1' else '0';
if IsExternalAbort(fault) then fsr<12> = fault.extflag;
fsr<9> = '1';
fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
```

```plaintext
return fsr;
```
Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```plaintext
// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
assert fault.statuscode != Fault_None;

bits(32) fsr = Zeros();
if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
if d_side then
    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
        fsr<13> = '1'; fsr<11> = '1';
    else
        fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain; // Domain field (data fault only)
return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```plaintext
// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
assert fault.statuscode != Fault_None;

bits(25) iss = Zeros();
if HaveRASExt() && IsAsyncAbort(fault) then
    iss<11:10> = fault.errortype; // AET
if d_side then
    if (IsSecondStage(fault) && !fault.s2fs1walk &&
        (!HaveRASExt() && fault.acctype == AccType_TTW &&
        boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) then
        iss<24:14> = LSInstructionSyndrome();
    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
        iss<8> = '1'; iss<6> = '1';
    else
        iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
return iss;
```
Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```c
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

bits(5) result;

case statuscode of
when Fault_AccessFlag
    assert level IN {1,2};
    result = if level == 1 then '00011' else '00110';
when Fault_Alignment
    result = '00001';
when Fault_Permission
    assert level IN {1,2};
    result = if level == 1 then '01101' else '01111';
when Fault_Domain
    assert level IN {1,2};
    result = if level == 1 then '01001' else '01011';
when Fault_Translation
    assert level IN {1,2};
    result = if level == 1 then '00101' else '00111';
when Fault_SyncExternal
    result = '01000';
when Fault_SyncExternalOnWalk
    assert level IN {1,2};
    result = if level == 1 then '01100' else '01110';
when Fault_SyncParity
    result = '11001';
when Fault_SyncParityOnWalk
    assert level IN {1,2};
    result = if level == 1 then '11100' else '11110';
when Fault_AsyncParity
    result = '11000';
when Fault_AsyncExternal
    result = '10110';
when Fault_Debug
    result = '00010';
when Fault_TLBConflict
    result = '10000';
when Fault_Lockdown
    result = '10100';  // IMPLEMENTATION DEFINED
when Fault_Exclusive
    result = '10101';  // IMPLEMENTATION DEFINED
when Fault_ICacheMaint
    result = '00100';
otherwise
    Unreachable();

return result;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm

```c
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

return imm32;
```
Library pseudocode for aarch32/functions/common/A32ExpandImm_C

// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

unrotated_value = ZeroExtend(imm12<7:0>, 32);
(imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);
return (imm32, carry_out);

Library pseudocode for aarch32/functions/common/DecodeImmShift

// DecodeImmShift()
// ================

(SRTYPE, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

SRTYPE shift_t;
integer shift_n;
case srtype of
  when '00' shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
  when '01' shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
  when '10' shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
  when '11' if imm5 == '00000' then
    shift_t = SRTYPE_RRX; shift_n = 1;
  else
    shift_t = SRTYPE_ROR; shift_n = UInt(imm5);
return (shift_t, shift_n);

Library pseudocode for aarch32/functions/common/DecodeRegShift

// DecodeRegShift()
// ================

SRTYPE DecodeRegShift(bits(2) srtype)

SRTYPE shift_t;
case srtype of
  when '00' shift_t = SRTYPE_LSL;
  when '01' shift_t = SRTYPE_LSR;
  when '10' shift_t = SRTYPE_ASR;
  when '11' shift_t = SRTYPE_ROR;
return shift_t;

Library pseudocode for aarch32/functions/common/RRX

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
(result, -) = RRX_C(x, carry_in);
return result;
Library pseudocode for aarch32/functions/common/RRX_C

// RRX_C()
// ========
(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

Library pseudocode for aarch32/functions/common/SRType

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};

Library pseudocode for aarch32/functions/common/Shift

// Shift()
// ========
<bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;

Library pseudocode for aarch32/functions/common/Shift_C

// Shift_C()
// =========
<bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType_RRX && amount != 1);
    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);
        return (result, carry_out);

Library pseudocode for aarch32/functions/common/T32ExpandImm

// T32ExpandImm()
// ==============
<bits(32) T32ExpandImm(bits(12) imm12)
    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);
    return imm32;
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
bits(32) imm32;
bit carry_out;
if imm12<11:10> == '00' then
  case imm12<9:8> of
    when '00'
      imm32 = ZeroExtend(imm12<7:0>, 32);
    when '01'
      imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
    when '10'
      imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
    when '11'
      imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
  carry_out = carry_in;
else
  unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
  (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));
return (imm32, carry_out);

Library pseudocode for aarch32/functions/common/VBitOps

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

Library pseudocode for aarch32/functions/common/VCGEType

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};

Library pseudocode for aarch32/functions/common/VCGTtype

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

Library pseudocode for aarch32/functions/common/VFPNegMul

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

// AArch32.CheckCP15InstrCoarseTraps()
// ========================================================
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) || (EL2Enabled() && !ELUsingAArch32(EL2))) then
    AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
    (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
    (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

// Check for coarse-grained Hyp traps
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    major = if nreg == 1 then CRn else CRm;
    // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
    // and MRC and MCR disabled by HCR.TIDCP.
    if (!(major IN {4,14}) && HSTR<major> == '1') ||
        (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding) then
        if (PSTATE.EL == EL0 &&
            boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
            UNDEFINED;
        if ELUsingAArch32(EL2) then
            AArch32.SystemAccessTrap(M32_Hyp, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

// AArch32.ExclusiveMonitorsPass()
// ==================================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);
    pasted = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());
    if pasted then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    return pasted;
Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual address region of size bytes starting at address.
// It is permitted (but not required) for this function to return FALSE and cause a store exclusive to fail if the virtual address region is not totally included within the region recorded by MarkExclusiveVA().
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);

Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);

Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

// AArch32.SetExclusiveMonitors()
// Sets the Exclusives monitors for the current PE to record the addresses associated with the virtual address region of size bytes starting at address.
AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    if IsFault(memaddrdesc) then
        return;
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
        AArch32.MarkExclusiveVA(address, ProcessorID(), size);

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

// CheckAdvSIMDEnabled()
// returns TRUE if Advanced SIMD access is permitted
CheckAdvSIMDEnabled()
    fpexc_check = TRUE;
    advsimd = TRUE;
    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
// Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted
    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];
    return;
Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```c
// CheckAdvSIMDOrVFPEnabled()
// --------------------------
CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrVFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```c
// CheckCryptoEnabled32()
// ---------------------
CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckVFPEnabled

```c
// CheckVFPEnabled()
// ----------------
CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/FPHalvedSub

```c
// FPHalvedSub()
// =============
bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FP_RoundingMode(fpcr);
    (type1,sign1,value1) = FP_Unpack(op1, fpcr);
    (type2,sign2,value2) = FP_Unpack(op2, fpcr);
    (done,result) = FP_ProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FP_Type_Infinity);
        inf2 = (type2 == FP_Type_Infinity);
        zero1 = (type1 == FP_Type_Zero);
        zero2 = (type2 == FP_Type_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FP_DefaultNaN(fpcr);
            FP_ProcessException(FP_Exc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FP_Rounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FP_Round(result_value, fpcr);
        end
    end
    return result;
```
Library pseudocode for aarch32/functions/float/FPRSqrtStep

// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
assert N IN {16,32};
  FPCRType fpcr = StandardFPSCRValue();
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    bits(N) product;
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero('0');
    else
      product = FPMul(op1, op2, fpcr);
  bits(N) three = FPTwo('0');
  result = FPHalvedSub(three, product, fpcr);
return result;

Library pseudocode for aarch32/functions/float/FPRecipStep

// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
assert N IN {16,32};
  FPCRType fpcr = StandardFPSCRValue();
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);
    bits(N) product;
    if (inf1 && zero2) || (zero1 && inf2) then
      product = FPZero('0');
    else
      product = FPMul(op1, op2, fpcr);
  bits(N) two = FPTwo('0');
  result = FPSub(two, product, fpcr);
return result;

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

// StandardFPSCRValue()
// =============

FPCRType StandardFPSCRValue()
//
bits(32) upper = '00000000000000000000000000000000';
bits(32) lower = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000000000000000';
return upper : lower;

Shared Pseudocode Functions
// AArch32.CheckAlignment()
// ========================
boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype, boolean iswrite)

bit A;
if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
    A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
elsif PSTATE.EL == EL2 then
    A = HSCTLR.A;
else
    A = SCTLR.A;
end;
aligned = (address == Align(address, alignment));
atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                    AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
                    AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
vector = acctype == AccType_VEC;
// AccType_VEC is used for SIMD element alignment checks only
check = (atomic || ordered || vector || A == '1');
if check && !aligned then
    secondstage = FALSE;
    AArch32.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
end;
return aligned;
// AArch32.MemSingle[] - non-assignment (read) form
// -----------------------------------------------
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] boolean ispair = FALSE;
return AArch32.MemSingle[address, size, acctype, aligned, ispair];

// AArch32.MemSingle[] - non-assignment (read) form
// -----------------------------------------------
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair]
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);
AddressDescriptor memaddrdesc;
bv(size*8) value;
iswrite = FALSE;
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);

PhysMemRetStatus memstatus;
(memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
  HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
return value;

// AArch32.MemSingle[] - assignment (write) form
// ----------------------------------------------

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value boolean ispair = FALSE;
return AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;

// AArch32.MemSingle[] - assignment (write) form
// ----------------------------------------------
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);
AddressDescriptor memaddrdesc;
iswrite = TRUE;
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);
// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
  ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
// Memory array access
accdesc = CreateAccessDescriptor(acctype);

memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
  HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
return;
Library pseudocode for aarch32/functions/memory/Hint_PreloadData

Hint_PreloadData(bits(32) address);

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

Hint_PreloadDataForWrite(bits(32) address);

Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

Hint_PreloadInstr(bits(32) address);

Library pseudocode for aarch32/functions/memory/MemA

// MemA[] - non-assignment form
// ============================
bits(8*size) MemA[bits(32) address, integer size]
  acctype = AccType_ATOMIC;
  return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ========================
MemA[bits(32) address, integer size] = bits(8*size) value
  acctype = AccType_ATOMIC;
  Mem_with_type[address, size, acctype] = value;
  return;

Library pseudocode for aarch32/functions/memory/MemO

// MemO[] - non-assignment form
// ============================
bits(8*size) MemO[bits(32) address, integer size]
  acctype = AccType_ORDERED;
  return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ========================
MemO[bits(32) address, integer size] = bits(8*size) value
  acctype = AccType_ORDERED;
  Mem_with_type[address, size, acctype] = value;
  return;

Library pseudocode for aarch32/functions/memory/MemS

// MemS[] - non-assignment form
// Memory accessor for streaming load multiple instructions
// ============================
bits(8*size) MemS[bits(32) address, integer size]
  acctype = AccType_A32LSMD;
  return Mem_with_type[address, size, acctype];

// MemS[] - assignment form
// Memory accessor for streaming store multiple instructions
// ========================
MemS[bits(32) address, integer size] = bits(8*size) value
  acctype = AccType_A32LSMD;
  Mem_with_type[address, size, acctype] = value;
  return;
Library pseudocode for aarch32/functions/memory/MemU

// MemU[] - non-assignment form
// ============================

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// =======================

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;

Library pseudocode for aarch32/functions/memory/MemU_unpriv

// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
Mem_with_type[] - non-assignment (read) form

Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
Instruction fetches would call AArch32.MemSingle directly.

```
bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype] = Mem_with_type[bits(32) address, integer size, AccType acctype]
  boolean ispair = FALSE;
  return Mem_with_type[address, size, acctype, ispair];

assert size IN {1, 2, 4, 8, 16};
constant halfsize = size DIV 2;
bits(size * 8) value;
boolean iswrite = FALSE;
boolean aligned;
if ispair then
  // check alignment on size of element accessed, not overall access size
  aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
else
  aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory access will generate an Alignment Fault, as to get this far means the first byte did not, so we must be changing to a new translation page.
  c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
  assert c IN {Constraint_FAULT, Constraint_NONE};
  if c == Constraint_NONE then aligned = TRUE;
for i = 1 to size-1
  value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
else
  value = AArch32.MemSingle[address, size, acctype, aligned, ispair];
if BigEndian(acctype) then
  value = BigEndianReverse(value);
return value;
```

Mem_with_type[] - assignment (write) form

```
Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value_in
  boolean ispair = FALSE;
  Mem_with_type[address, size, acctype, ispair] = value_in;

assert size IN {1, 2, 4, 8, 16};
constant halfsize = size DIV 2;
bits(size*8) value = value_in;
boolean iswrite = TRUE;
boolean aligned;
if BigEndian(acctype) then
  value = BigEndianReverse(value);
if ispair then
  // check alignment on size of element accessed, not overall access size
  aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
else
  aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
if !aligned then
  assert size > 1;
  AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
assert c IN {Constraint_FAULT, Constraint_NONE};
if c == Constraint_NONE then aligned = TRUE;

for i = 1 to size-1
    AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
else
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

// AArch32.ESBOperation()
// =============
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
    AArch64.ESBOperation();
    return;

route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');

bits(5) target;
if route_to_monitor then
    target = M32_Monitor;
elsif route_to_hyp || PSTATE.M == M32_Hyp then
    target = M32_Hyp;
else
    target = M32_Abort;

boolean mask_active;
if IsSecure() then
    mask_active = TRUE;
elsif target == M32_Monitor then
    mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
    mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32(target);
intdis = Halted() || ExternalDebugInterruptsDisabled(el);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending() then
    syndrome32 = AArch32.PhysicalSErrorSyndrome();
    DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    ClearPendingPhysicalSError();
return;

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

```c
// AArch32.ReportDeferredSError()
// ==============================
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
```

```c
bits(32) target;
target<31> = '1'; // A
syndrome = Zeros(16);
if PSTATE.EL == EL2 then
    syndrome<11:10> = AET; // AET
    syndrome<9> = ExT; // EA
    syndrome<5:0> = '010001'; // DFSC
else
    syndrome<15:14> = AET; // AET
    syndrome<12> = ExT; // ExT
    syndrome<9> = TTBCR.EAE; // LPAE
    if TTBCR.EAE == '1' then // Long-descriptor format
        syndrome<5:0> = '010001'; // STATUS
    else // Short-descriptor format
        syndrome<10,3:0> = '10110'; // FS
if HaveAArch64() then
    target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
else
    target<15:0> = syndrome;
return target;
```

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```c
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```c
// AArch32.vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
```

```c
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

// Check for EL2 using AArch64 state
if !ELUsingAArch32(EL2) then
    AArch64.vESBOperation();
return;

// If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a virtual SError interrupt might be pending
vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
vSEI_pending = vSEI_enabled && HCR.VA == '1';
vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked = vintdis || PSTATE.A == '1';

// Check for a masked virtual SError pending
if vSEI pending && vmasked then
    VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
    HCR.VA = '0'; // Clear pending virtual SError
return;
```
Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

// AArch32.ResetGeneralRegisters()
// ----------------------------------------

AArch32.ResetGeneralRegisters()
for i = 0 to 7
    R[i] = bits(32) UNKNOWN;
for i = 8 to 12
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;   // No R14_hyp
for i = 13 to 14
    Rmode[i, M32_User] = bits(32) UNKNOWN;
    Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
    Rmode[i, M32_Svc] = bits(32) UNKNOWN;
    Rmode[i, M32_Abort] = bits(32) UNKNOWN;
    Rmode[i, M32_Undef] = bits(32) UNKNOWN;
if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;
return;

Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

// AArch32.ResetSIMDFPRegisters()
// ------------------------------

AArch32.ResetSIMDFPRegisters()
for i = 0 to 15
    Q[i] = bits(128) UNKNOWN;
return;

Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

// AArch32.ResetSpecialRegisters()
// --------------------------------

AArch32.ResetSpecialRegisters()
// AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSRund<31:0> = bits(32) UNKNOWN;
if HaveEL(EL2) then
    SPSR_hyp = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
if HaveEL(EL3) then
    SPSR_mon = bits(32) UNKNOWN;
// External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
return;

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

AArch32.ResetSystemRegisters(boolean cold_reset);
Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

// ALUExceptionReturn()
// ==============

ALUExceptionReturn(bits(32) address)
if PSTATE.EL == EL2 then
  UNDEFINED;
elsif PSTATE.M IN {M32_User, M32_System} then
  Constraint c = ConstrainUnpredictable(Unpredictable_ALU_EXCEPTION_RETURN);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
  when Constraint_UNDEF
    UNDEFINED;
  when Constraint_NOP
    EndOfInstruction();
else
  AArch32.ExceptionReturn(address, SPSR[]);

Library pseudocode for aarch32/functions/registers/ALUWritePC

// ALUWritePC()
// ===========

ALUWritePC(bits(32) address)
if CurrentInstrSet() == InstrSet_A32 then
  BXWritePC(address, BranchType_INDIR);
else
  BranchWritePC(address, BranchType_INDIR);

Library pseudocode for aarch32/functions/registers/BXWritePC

// BXWritePC()
// ===========

BXWritePC(bits(32) address_in, BranchType branch_type)
bits(32) address = address_in;
if address<0> == '1' then
  SelectInstrSet(InstrSet_T32);
  address<0> = '0';
else
  SelectInstrSet(InstrSet_A32);
  // For branches to an unaligned PC counter in A32 state, the processor takes the branch
  // and does one of:
  // * Forces the address to be aligned
  // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
  if address<1> == '1' \&\& ConstrainUnpredictableBool(Unpredictable_A32_FORCEALIGNPC) then
    address<1> = '0';
  boolean branch_conditional = AArch32.CurrentCond() != '111x';
  BranchTo(address, branch_type, branch_conditional);

Library pseudocode for aarch32/functions/registers/BranchWritePC

// BranchWritePC()
// ===============

BranchWritePC(bits(32) address_in, BranchType branch_type)
bits(32) address = address_in;
if CurrentInstrSet() == InstrSet_A32 then
  address<1:0> = '00';
else
  address<0> = '0';
boolean branch_conditional = AArch32.CurrentCond() != '111x';
BranchTo(address, branch_type, branch_conditional);
// CBWritePC()
// ===========
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
   bits(32) address = address_in;
   assert CurrentInstrSet() == InstrSet_T32;
   address<0> = '0';
   boolean branch_conditional = TRUE;
   BranchTo(address, BranchType_DIR, branch_conditional);

// D[] - non-assignment form
// --------------------------

bits(64) D[n] = bits(64) value
   assert n >= 0 && n <= 31;
   base = (n MOD 2) * 64;
   bits(128) vreg = V[n DIV 2];
   vreg<base+63:base> = value;
   V[n DIV 2] = vreg;
   return;

// D[] - assignment form
// -----------------------

D[n] = bits(64) value
   assert n >= 0 && n <= 31;
   base = (n MOD 2) * 64;
   bits(128) vreg = V[n DIV 2];
   vreg<base+63:base> = value;
   V[n DIV 2] = vreg;
   return;

// Din[] - non-assignment form
// -----------------------------

bits(64) Din[n] = _Dclone[n];

// LR - assignment form
// ---------------------

LR = bits(32) value
   R[14] = value;
   return;

// LR - non-assignment form
// -------------------------

bits(32) LR
   return R[14];

// LoadWritePC()
// ===============

LoadWritePC(bits(32) address)
   BXWritePC(address, BranchType_INDIR);
// LookUpRIndex()
// ===============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of // Select index by mode:    usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

bits(32) SP_mon;
bites(32) LR_mon;

Library pseudocode for aarch32/functions/registers/PC

// PC - non-assignment form
// ===============

bits(32) PC
    return R[15]; // This includes the offset from AArch32 state

Library pseudocode for aarch32/functions/registers/PCStoreValue

// PCStoreValue()
// ===============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;

Library pseudocode for aarch32/functions/registers/Q

// Q[] - non-assignment form
// ===============

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n];

// Q[] - assignment form
// ===============

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n] = value;
    return;
Library pseudocode for aarch32/functions/registers/Qin

// Qin[] - non-assignment form
// ================

bits(128) Qin[integer n]
assert n >= 0 && n <= 15;
return Din[2*n+1]:Din[2*n];

Library pseudocode for aarch32/functions/registers/R

// R[] - assignment form
// ==============

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// ================

bits(32) R[integer n]
if n == 15 then
    offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
    return _PC<31:0> + offset;
else
    return Rmode[n, PSTATE.M];

Library pseudocode for aarch32/functions/registers/RBankSelect

// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq, integer svc, integer abt, integer und, integer hyp)

    integer result;
    case mode of
        when M32_User result = usr; // User mode
        when M32_FIQ result = fiq; // FIQ mode
        when M32_IRQ result = irq; // IRQ mode
        when M32_Svc result = svc; // Supervisor mode
        when M32_Abort result = abt; // Abort mode
        when M32_Hyp result = hyp; // Hyp mode
        when M32_Undef result = und; // Undefined mode
        when M32_System result = usr; // System mode uses User mode registers
        otherwise Unreachable(); // Monitor mode
    return result;
// Rmode[] - non-assignment form
// ------------------------------------------

bits(32) Rmode[integer n, bits(5) mode]
assert n >= 0 & n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
  if n == 13 then return SP_mon;
  elsif n == 14 then return LR_mon;
  else return _R[n]<31:0>;
else
  return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// ------------------------------------------

Rmode[integer n, bits(5) mode] = bits(32) value
assert n >= 0 & n <= 14;

// Check for attempted use of Monitor mode in Non-secure state.
if !IsSecure() then assert mode != M32_Monitor;
assert !BadMode(mode);

if mode == M32_Monitor then
  if n == 13 then SP_mon = value;
  elsif n == 14 then LR_mon = value;
  else _R[n]<31:0> = value;
else
  // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
  // register are unchanged or set to zero. This is also tested for on
  // exception entry, as this applies to all AArch32 registers.
  if HaveAAArch64() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
    _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
  else
    _R[LookUpRIndex(n, mode)]<31:0> = value;

return;

// S[] - non-assignment form
// ------------------------------------------

bits(32) S[integer n]
assert n >= 0 & n <= 31;
base = (n MOD 4) * 32;
bits(128) vreg = V[n DIV 4];
return vreg<base+31:base>;

// S[] - assignment form
// ------------------------------------------

S[integer n] = bits(32) value
assert n >= 0 & n <= 31;
base = (n MOD 4) * 32;
bits(128) vreg = V[n DIV 4];
vreg<base+31:base> = value;
V[n DIV 4] = vreg;
return;
Library pseudocode for aarch32/functions/registers/SP

// SP - assignment form
// ====================
SP = bits(32) value
    R[13] = value;
    return;

// SP - non-assignment form
// ========================
bits(32) SP
    return R[13];

Library pseudocode for aarch32/functions/registers/_Dclone

array bits(64) _Dclone[0..31];

Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

// AArch32.ExceptionReturn()
// =========================
AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();
    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';  // T32
        else
            new_pc<1:0> = '00';  // A32
    boolean branch_conditional = AArch32.CurrentCond() != '111x';
    BranchTo(new_pc, BranchType_ERET, branch_conditional);
    CheckExceptionCatch(FALSE);  // Check for debug event on exception return

Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

// AArch32.ExecutingCP10or11Instr()
// ================================
boolean AArch32.ExecutingCP10or11Instr()
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN (InstrSet_A32, InstrSet_T32);
    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    else  // InstrSet_T32
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```c
// AArch32.ITAdvance()
// ============

AArch32.ITAdvance()
if PSTATE.IT<2:0> == '000' then
    PSTATE.IT = '00000000';
else
    PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
return;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```c
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].
AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```c
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].
AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```c
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
assert UsingAArch32();
assert (cp_num IN {14,15});
assert cp_num == UInt(instr<11:8>);

opc1 = UInt(instr<23:21>);
opc2 = UInt(instr<7:5>);
CRn = UInt(instr<19:16>);
CRm = UInt(instr<3:0>);

if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
    return TRUE;
else
    return FALSE;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```c
// Read the contents of R[t] and write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```c
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```c
// Read a value from a virtual address and write it to an AArch32 System register.
AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```
Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```c
// AArch32.WriteMode()
// ===============
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.
AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFFromM32(mode);
    assert valid;
    PSTATE.M = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```c
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.II.
AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFFromM32(mode);
    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;
    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;
    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;
    if !valid then
        PSTATE.II = '1';
    else
        AArch32.WriteModeByInstr();
```

Shared Pseudocode Functions
Library pseudocode for aarch32/functions/system/BadMode

// BadMode()
// =========
boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
boolean valid;
case mode of
  when M32_Monitor
    valid = HaveAArch32EL(EL3);
  when M32_Hyp
    valid = HaveAArch32EL(EL2);
  when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
    // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
    // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
    // AArch64, then these modes are EL1 modes.
    // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
    valid = HaveAArch32EL(EL1);
  when M32_User
    valid = HaveAArch32EL(EL0);
  otherwise
    valid = FALSE;       // Passed an illegal mode value
return !valid;

Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.
BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
  when '000xx', '00100'                          // R8_usr to R12_usr
    if mode != M32_FIQ then UNPREDICTABLE;
  when '00101'                                   // SP_usr
    if mode == M32_System then UNPREDICTABLE;
  when '00110'                                   // LR_usr
    if mode IN {M32_Hyp, M32_System} then UNPREDICTABLE;
  when '010xx', '0110x', '01110'                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
    if mode == M32_FIQ then UNPREDICTABLE;
  when '1000x'                                   // LR_irq, SP_irq
    if mode == M32_IRQ then UNPREDICTABLE;
  when '1001x'                                   // LR_svc, SP_svc
    if mode == M32_Svc then UNPREDICTABLE;
  when '1010x'                                   // LR_abt, SP_abt
    if mode == M32_Abort then UNPREDICTABLE;
  when '1011x'                                   // LR_und, SP_und
    if mode == M32_Undef then UNPREDICTABLE;
  when '1110x'                                   // LR_mon, SP_mon
    if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
  when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
    if !HaveEL(EL2) || (mode IN {M32_Monitor, M32_Hyp}) then UNPREDICTABLE;
  when '11111'                                   // SP_hyp, only from Monitor mode
    if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
  otherwise
    UNPREDICTABLE;
return;
// CPSRWriteByInstr()
// ===============

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>; // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        if privileged then
            PSTATE.E = value<9>; // PSTATE.E is writable at EL0

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
    return;

Library pseudocode for aarch32/functions/system/ConditionPassed

// ConditionPassed()
// ================

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());

Library pseudocode for aarch32/functions/system/CurrentCond

bits(4) AArch32.CurrentCond();

Library pseudocode for aarch32/functions/system/InITBlock

// InITBlock()
// =============

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
Library pseudocode for aarch32/functions/system/LastInITBlock

// LastInITBlock()
// ===============
boolean LastInITBlock()
  return (PSTATE.IT<3:0> == '1000');

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

// SPSRWriteByInstr()
// ==================
SPSRWriteByInstr(bits(32) value, bits(4) bytemask)
  bits(32) new_spsr = SPSR[];
  if bytemask<3> == '1' then
    new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits
  if bytemask<2> == '1' then
  if bytemask<1> == '1' then
    new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask
  if bytemask<0> == '1' then
    new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits
  SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode
  return;

Library pseudocode for aarch32/functions/system/SPSRaccessValid

// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE
SPSRaccessValid(bits(5) SYSm, bits(5) mode)
  case SYSm of
    when '01110'                                                   // SPSR_fiq
      if mode == M32_FIQ then UNPREDICTABLE;
    when '10000'                                                   // SPSR_irq
      if mode == M32_IRQ then UNPREDICTABLE;
    when '10010'                                                   // SPSR_svc
      if mode == M32_Svc then UNPREDICTABLE;
    when '10100'                                                   // SPSR_abt
      if mode == M32_Abort then UNPREDICTABLE;
    when '10110'                                                   // SPSR_und
      if mode == M32_Undef then UNPREDICTABLE;
    when '11100'                                                   // SPSR_mon
      if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
    when '11110'                                                   // SPSR_hyp
      if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
    otherwise
      UNPREDICTABLE;
  return;
Library pseudocode for aarch32/functions/system/SelectInstrSet

// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};
    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
    return;

Library pseudocode for aarch32/functions/v6simd/Sat

// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;

Library pseudocode for aarch32/functions/v6simd/SignedSat

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
Library pseudocode for aarch32/ic/AArch32.IC
// AArch32.IC()
// ============
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
regval = bits(32) UNKNOWN;
AArch32.IC(regval, opscope);

// AArch32.IC()
// ============
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
  CacheRecord cache;
  AccType accctype = AccType_IC;
  cache.acctype = acctype;
  cache.cachetype = CacheType_Instruction;
  cache.cacheop = CacheOp_Invalidate;
  cache.opscope = opscope;
  cache.security = SecurityStateAtEL(PSTATE.EL);
  if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
    if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1 && EL2Enabled() && HCR.FB == '1') then
      cache.shareability = Shareability_ISH;
    else
      cache.shareability = Shareability_NSH;
      cache.regval = ZeroExtend(regval);
      CACHE_OP(cache);
    else
assert opscope == CacheOpScope_PoU;
    if EL2Enabled() then
      if PSTATE.EL IN {EL0, EL1} then
        cache.is_vmid_valid = TRUE;
        cache.vmid = VMID[];
      else
        cache.is_vmid_valid = FALSE;
      else
        cache.is_vmid_valid = FALSE;
    if PSTATE.EL == EL0 then
      cache.is_asid_valid = TRUE;
      cache.asid = ASID[];
    else
      cache.is_asid_valid = FALSE;
    need_translate = ICInstNeedsTranslation(opscope);
    cache.shareability = Shareability_NSH;
    cache.vaddress = ZeroExtend(regval);
    cache.translated = need_translate;
    if !need_translate then
      cache.paddress = FullAddress UNKNOWN;
      CACHE_OP(cache);
      return;
    wasaligned = TRUE;
    iswrite = FALSE;
    size = 0;
    memaddrdesc = AArch32.TranslateAddress(regval, acctype, iswrite, wasaligned, size);
    if IsFault(memaddrdesc) then
      AArch32.Abort(regval, memaddrdesc.fault);
    cache.paddress = memaddrdesc.paddress;
    CACHE_OP(cache);
    return;
// RestrictPrediction()
// Clear all predictions in the context.
AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

ExecutionCntxt c;
target_el = val<25:24>;

// If the instruction is executed at an EL lower than the specified
// level, it is treated as a NOP.
if UInt(target_el) > UInt(PSTATE.EL) then return;

bit ns = val<26>;
ss = TargetSecurityState(ns);
c.security = ss;
c.target_el = target_el;

if EL2Enabled() then
    if PSTATE.EL IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid = FALSE;
        c.vmid = VMID[];
    elsif target_el IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid = val<27> == '1';
        c.vmid = ZeroExtend(val<23:16>, 16); // Only valid if val<27> == '0';
    else
        c.is_vmid_valid = FALSE;
    end
else
    c.is_vmid_valid = FALSE;
end

if PSTATE.EL == EL0 then
    c.is_asid_valid = TRUE;
    c.all_asid = FALSE;
    c.asid = ASID[];
elsif target_el == EL0 then
    c.is_asid_valid = TRUE;
    c.all_asid = val<8> == '1';
    c.asid = ZeroExtend(val<7:0>, 16); // Only valid if val<8> == '0';
else
    c.is_asid_valid = FALSE;
end

c.restriction = restriction;
RESTRICT_PREDICTIONS(c);
Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode
MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit B_in, bit S)

memattrs;

bits(3) TEX = TEX_in;
bit C = C_in;
bit B = B_in;

// Reserved values map to allocated values
if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
  bits(5) texcb;
(-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB);
TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;

// Distinction between Inner Shareable and Outer Shareable is not supported in this format
// A memory region is either Non-shareable or Outer Shareable

case TEX:C:B of
  when '00000'
    // Device-nGnRnE
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
  when '00001', '01000'
    // Device-nGnRE
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRE;
    memattrs.shareability = Shareability_OSH;
  when '00010'
    // Write-through Read allocate
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WT;
    memattrs.inner.hints = MemHint_RA;
    memattrsouter.attrs = MemAttr_WT;
    memattrs.outer.hints = MemHint_RA;
    memattrs.shareability = if S == '1' then Shareability_OSH else Shareability_NSH;
  when '00011'
    // Write-back Read allocate
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RA;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RA;
    memattrs.shareability = if S == '1' then Shareability_OSH else Shareability_NSH;
  when '00100'
    // Non-cacheable
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_NC;
    memattrs.outer.attrs = MemAttr_NC;
    memattrs.shareability = Shareability_OSH;
  when '00110'
    memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
  when '00111'
    // Write-back Read and Write allocate
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.inner.hints = MemHint_RWA;
    memattrs.outer.attrs = MemAttr_WB;
    memattrs.outer.hints = MemHint_RWA;
    memattrs.shareability = if S == '1' then Shareability_OSH else Shareability_NSH;
  when '1xxxx'
    // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
    memattrs.memtype = MemType_Normal;
    memattrs.inner = DecodeSDFAttr(C:B);
    memattrs.outer = DecodeSDFAttr(TEX<1:0>);
    if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
      memattrs.shareability = Shareability_OSH;
    else
      // Shared Pseudocode Functions

Shared Pseudocode Functions
memattrs.shareability = if S == '1' then Shareability_OSH else Shareability_NSH;
otherwise
    // Reserved, handled above
    Unreachable();

// The Transient hint is not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;
memattrs.tagged          = FALSE;

if memattrs.inner.attrs == MemAttr_WR && memattrs.outer.attrs == MemAttr_WR then
    memattrs.xs = '0';
else
    memattrs.xs = '1';

return memattrs;
Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

// AArch32.RemappedTEXDecode()
// ===========================
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit C, bit B, bit S)

    MemoryAttributes memattrs;
    PRRR_Type prrr;
    NMRR_Type nmrr;

    region = UInt(TEX<0>:C:B);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        return MemoryAttributes IMPLEMENTATION_DEFINED;
    if regime == Regime_EL30 then
        prrr = PRRR_S;
        nmrr = NMRR_S;
    elsif HaveAArch32EL(EL3) then
        prrr = PRRR_NS;
        nmrr = NMRR_NS;
    else
        prrr = PRRR;
        nmrr = NMRR;

    base = 2 * region;
    attrfield = prrr<base+1:base>;

    if attrfield == '11' then      // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRRR);
    case attrfield of
        when '00'                  // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '01'                  // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '10'
            NSn = if S == '0' then prrr.NS0 else prrr.NS1;
            NOSm = prrr<region+24> AND NSn;
            IRn = nmrr<base+1:base>;
            ORn = nmrr<base+17:base+16>;
            memattrs.memtype = MemType_Normal;
            memattrs.inner = DecodeSDFAttr(IRn);
            memattrs.outer = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                bits(2) sh = NSn:NOSm;
                memattrs.shareability = DecodeShareability(sh);
        when '11'
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged = FALSE;

    if memattrs.inner.attrs == MemAttr_WR && memattrs.outer.attrs == MemAttr_WR then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
assert FLUsingAArch32(S1TranslationRegime());
assert size IN {2,4};

match = FALSE;
mismatch = FALSE;
for i = 0 to NumBreakpointsImplemented() - 1
  (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
  match = match || match_i;
  mismatch = mismatch || mismatch_i;
if match && HaltOnBreakpointOrWatchpoint() then
  reason = DebugHalt_Breakpoint;
  Halt(reason);
elsif (match || mismatch) then
  acctype = AccType_IFETCH;
  iswrite = FALSE;
  debugmoe = DebugException_Breakpoint;
  return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
  return NoFault();

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

FaultRecord fault = NoFault();
d_side = (acctype != AccType_IFETCH);
generate_exception = AArch32.GenerateDebugExceptions() && DBGSCR.MDBGen == '1';
halt = HaltOnBreakpointOrWatchpoint();
// Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);
if !d_side && vector_catch_first && generate_exception then
  fault = AArch32.CheckVectorCatch(vaddress, size);
if fault.statuscode == Fault_None && (generate_exception || halt) then
  if d_side then
    fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
  else
    fault = AArch32.CheckBreakpoint(vaddress, size);
if fault.statuscode == Fault_None && !d_side && !vector_catch_first && generate_exception then
  return AArch32.CheckVectorCatch(vaddress, size);
return fault;
Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

// AArch32.CheckVectorCatch()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());

match = AArch32.VCMatch(vaddress);
if size == 4 && !match && AArch32.VCMatch(vaddress + 2) then
    match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

if match then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    debugmoe = DebugException_VectorCatch;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return NoFault();

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());

if acctype IN {AccType_TTW, AccType_IC, AccType_AT, AccType_ATPAN} then
    return NoFault();
if acctype == AccType_DC then
    if !iswrite then
        return NoFault();
    elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
        return NoFault();
    match = FALSE;
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;
    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress);
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();

Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

// AArch32.DebugFault()
// ================
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)
{
  FaultRecord fault;
  fault.statusCode = Fault_Debug;
  fault.acctype = acctype;
  fault.write = iswrite;
  fault.debugmoe = debugmoe;
  fault.secondstage = FALSE;
  fault.s2fs1walk = FALSE;
  return fault;
}

Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

// AArch32.IPAIsOutOfRange()
// =========================
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
{
  // Input Address size
  iasize = AArch32.S2IASize(walkparams.t0sz);
  return iasize < 40 && !IsZero(ipa<39:iasize>);
}

Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

// AArch32.S1HasAlignmentFault()
// =============================
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccType acctype, boolean aligned, bit ntlsmd, MemoryAttributes memattrs)
{
  if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
    return FALSE;
  if acctype == AccType_A32LSMD && ntlsmd == '0' && memattrs.device != DeviceType_GRE then
    return TRUE;
  return !aligned || acctype == AccType_DCZVA;
}
Library pseudocode for aarch32/translation/faults/AArch32.S1LDHasPermissionsFault
// AArch32.S1LDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, SecurityState ss, S1TTWParams walkparams,
Permissions perms, MemType memtype, PASpace paspace,
boolean ispriv, AccType acctype, boolean iswrite)

bit r;
bit w;
bit x;
bit pr;
bit pw;
bit ur;
bit uw;
if HasUnprivileged(regime) then
  // Apply leaf permissions
  case perms.ap<2:1> of
    when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
    when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
    when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
    when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
  // Apply hierarchical permissions
  case perms.ap_table of
    when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
    when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
    when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
    when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access
  xn = perms.xn OR perms.xn_table;
  pxn = perms.pxn OR perms.pxn_table;
  ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
  px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));
  pan.access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
  if HavePANExt() & pan.access then
    pan = PSTATE.PAN AND (ur OR uw);
    pr  = pr AND NOT(pan);
    pw  = pw AND NOT(pan);
  (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
  // Prevent execution from Non-secure space by PE in Secure state if SIF is set
  if ss == SS_Secure & paspace == PAS_NonSecure then
    x = x AND NOT(walkparams.sif);
  else
    // Apply leaf permissions
    case perms.ap<2:1> of
      when '0' (r,w) = ('1','1'); // No effect
      when '1' (r,w) = ('1','0'); // Read-only
    // Apply hierarchical permissions
    case perms.ap_table<1> of
      when '0' (r,w) = ( r , w ); // No effect
      when '1' (r,w) = ( r , '0'); // Read-only
    xn = perms.xn OR perms.xn_table;
    x = NOT(xn OR (w AND walkparams.wxn));
    if acctype == AccType_IFETCH then
      constraint = ConstrainUnpredictable(Unpredictable_INTRDEVICE);
    if constraint == Constraint_FAULT & memtype == MemType_Device then
      return TRUE;
    else
      return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
      return FALSE;
    else
      iswrite then
return w == '0';
else
    return r == '0';
Library pseudocode for aarch32/translation/faults/AArch32.S1SDHasPermissionsFault
// AArch32.S1SDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, SecurityState ss, Permissions perms_in, MemType memtype, PASpace paspace, boolean ispriv, AccType acctype, boolean iswrite)

Permissions perms = perms_in;
bit pr;
bit pw;
bit ur;
bit uw;
SCTLR_Type sctlr;
if regime == Regime_EL30 then
  sctlr = SCTLR_S;
elsif HaveAArch32EL(EL3) then
  sctlr = SCTLR_NS;
else
  sctlr = SCTLR;
if sctlr.AFE == '0' then
  // Map Reserved encoding '100'
  if perms.ap == '100' then
    perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";
  case perms.ap of
    when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
    when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
    when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
    when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
      '100' is reserved
    when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
    when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
    when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
  else // Simplified access permissions model
    case perms.ap<2:1> of
      when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
      when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
      when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
      when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.WXN));
  pan access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
  if HavePANExt() && pan access then
    pan = PSTATE.PAN AND (ur OR uw);
    pr  = pr AND NOT(pan);
    pw  = pw AND NOT(pan);
  (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
  // Prevent execution from Non-secure space by PE in Secure state if SIF is set
  if ss == SS_Secure && paspace == PAS_NonSecure then
    x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);
  if acctype == AccType_IFETCH then
    constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    if constraint == Constraint_FAULT && memtype == MemType_Device then
      return TRUE;
    else
      return x == '0';
  elsif acctype IN {AccType_IC, AccType_DC} then
    return FALSE;
  elsif iswrite then
    return w == '0';
  else
    return r == '0';

Shared Pseudocode Functions
Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

// AArch32.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccType acctype, boolean aligned, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;
    return !aligned || acctype == AccType_DCZVA;

Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

// AArch32.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(boolean s2fs1walk, S2TTWParams walkparams, Permissions perms, MemType memtype, boolean ispriv, AccType acctype, boolean iswrite)
    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xnx of
            when '00'  (px, ux) = ( r , r );
            when '01'  (px, ux) = ('0', r );
            when '10'  (px, ux) = ('0','0');
            when '11'  (px, ux) = ( r ,'0');
        x = if ispriv then px else ux;
    else
        x = r AND NOT(perms.s2xn);
    if s2fs1walk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';

Shared Pseudocode Functions
Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```java
// AArch32.S2InconsistentSL()  
// ---------------------------
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams) {
    startlevel = AArch32.S2StartLevel(walkparams.sl0);
    levels = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (levels * stride // Bits resolved by table walk, except initial level
                     + granulebits // Bits directly mapped to output address
                     + 1);           // At least 1 more bit to be decoded by initial level

    // Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;

    // Configured Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
}
```

Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```java
// AArch32.VAIsOutOfRange()  
// -------------------------
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va) {
    if regime == Regime_EL2 then
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
    end
}
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

```java
// AArch32.GetS1TLBContext()  
// ---------------------------
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va) {
    TLBContext tlbcontext;
    case regime of
        when Regime_EL2 tlbcontext = AArch32.TLBContextEL2(va);
        when Regime_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime_EL30 tlbcontext = AArch32.TLBContextEL30(va);
    end
    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
}
```
// AArch32.GetS2TLBContext()
// ================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address);
    tlbcontext.cnp         = if HaveCommonNotPrivateTransExt() then VTTBR.CnP else '0';

return tlbcontext;
// AArch32.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime
// (P10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
TLBContext tlbcontext;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;

if HaveAArch32EL(EL3) then
    ttbcr = TTBCR_NS;
    ttbr0 = TTBR0_NS;
    ttbr1 = TTBR1_NS;
    contextidr = CONTEXTIDR_NS;
else
    ttbcr = TTBCR;
    ttbr0 = TTBR0;
    ttbr1 = TTBR1;
    contextidr = CONTEXTIDR;

tlbcontext.ss = ss;
tlbcontext.regime = Regime_EL10;
if AArch32.EL2Enabled(ss) then
    tlbcontext.vmid = ZeroExtend(VTTBR.VMID);
if ttbcr.EAE == '1' then
    tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID);
else
    tlbcontext.asid = ZeroExtend(contextidr.ASID);

tlbcontext.tg = TGx_4KB;
tlbcontext.ia = ZeroExtend(va);
if HaveCommonNotPrivateTransExt() && ttbcr.EAE == '1' then
    if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VRange_LOWER then
        tlbcontext.cnp = ttbr0.CnP;
    else
        tlbcontext.cnp = ttbr1.CnP;
else
    tlbcontext.cnp = '0';

return tlbcontext;

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL2

// AArch32.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
TLBContext tlbcontext;

tlbcontext.ss = SS_NonSecure;
tlbcontext.regime = Regime_EL2;
tlbcontext.ia = ZeroExtend(va);

return tlbcontext;
Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL30

```c
// AArch32.TLBContextEL30()
// ========================
// Gather translation context for accesses under EL30 regime
// (P10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;
    tlbcontext.ss = SS_Secure;
    tlbcontext.regime = Regime_EL30;
    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID);
    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va);
    if HaveCommonNotPrivateTransExt() && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
        else
            tlbcontext.cnp = '0';
    return tlbcontext;
```

Library pseudocode for aarch32/translation/translation/AArch32.AccessUsesEL

```c
// AArch32.AccessUsesEL()
// ======================
// Determine the privilege associated with the access

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```c
// AArch32.EL2Enabled()
// ====================
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS_Secure then
        if !(HaveEL(EL2) & HaveSecureEL2Ext()) then return FALSE;
        elseif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```
Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

// AArch32.FullTranslate()
// =======================
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccType acctype, boolean iswrite, boolean aligned)

// Prepare fault fields in case a fault is detected
fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

regime = TranslationRegime(PSTATE.EL, acctype);
ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;
ss = SecurityStateForRegime(regime);

// First Stage Translation
AddressDescriptor ipa;
if regime == Regime_EL2 || TTBCR.EAE == '1' then
  (fault, ipa) = AArch32.S1TranslateLD(fault, regime, ss, va, acctype,
    aligned, iswrite, ispriv);
else
  (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, ss, va, acctype,
    aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
  return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);
if regime == Regime_EL10 && EL2Enabled() then
  ipa.vaddress = ZeroExtend(va);
s2fs1walk = FALSE;
AddressDescriptor pa;
  (fault, pa) = AArch32.S2Translate(fault, ipa, ss, s2fs1walk, acctype,
    aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
  return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);
else
  return pa;
else
  return ipa;

Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

// AArch32.OutputDomain()
// ========================
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
  bits(2) Dn;
  index = 2 * UInt(domain);
  if regime == Regime_EL30 then
    Dn = DACR_S<index+1:index>;
  elsif HaveAArch32EL(EL3) then
    Dn = DACR_NS<index+1:index>;
  else
    Dn = DACR<index+1:index>;
  if Dn == '10' then
    // Reserved value maps to an allocated value
    (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR);
  return Dn;
Library pseudocode for aarch32/translation/translation/AArch32.S1DisabledOutput
// AArch32.S1DisabledOutput()
// ------------------------
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime, SecurityState ss, bits(32) va, AccType acctype, boolean aligned)

FaultRecord fault = fault_in;
// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
bit default_cacheable;
if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
  if ELStateUsingAArch32(EL2, ss == SS_Secure) then
    default_cacheable = HCR.DC;
  else
    default_cacheable = HCR_EL2.DC;
else
  default_cacheable = '0';
if default_cacheable == '1' then
  // Use default cacheable settings
  memattrs.memtype = MemType_Normal;
  memattrs.inner.attrs = MemAttr_WB;
  memattrs.inner.hints = MemHint_RWA;
  memattrs.outer.attrs = MemAttr_WB;
  memattrs.outer.hints = MemHint_RWA;
  memattrs.shareability = Shareability_NS;
  if ELStateUsingAArch32(EL2, ss == SS_Secure) && HaveMTE2Ext() then
    memattrs.tagged = HCR_EL2.DCT == '1';
  else
    memattrs.tagged = FALSE;
else
  if acctype == AccType_IFETCH then
    memattrs.memtype = MemType_Normal;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;
  else
    memattrs.inner.attrs = MemAttr_NC;
    memattrs.outer.attrs = MemAttr_NC;
else
  // Treat memory region as Device
  memattrs.memtype = MemType_Device;
  memattrs.device = DeviceType_nGnRnE;
  memattrs.shareability = Shareability_OSH;
  memattrs.tagged = FALSE;

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
  case regime of
  when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
  when Regime_EL2 ntlsmd = HSCTLR.nTLSMD;
  when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
else
  ntlsmd = '1';
if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, memattrs) then
  fault.statuscode = Fault_Alignment;
  return (fault, AddressDescriptor UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va);
oa.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);
return (fault, ipa);

Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

// AArch32.S1Enabled()
// ===================
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    else
        return HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';
// AArch32.S1TranslateLD()
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime, SecurityState ss, bits(32) va, AccType acctype, boolean aligned, boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fs1walk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    return AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
walkparams = AArch32.GetS1TTWParams(regime, va);

if AArch32.VAIsOutOfRange(regime, walkparams, va) then
    fault.level = 1;
    fault.statuscode = Fault_Translation;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkLD(fault, regime, ss, walkparams, va, ispriv);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

if AArch32.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsm, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    if AArch32.S1LDHasPermissionsFault(regime, ss, walkparams, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, FALSE) then
        fault.statuscode = Fault_Permission;
fault.write = FALSE;
elseif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, TRUE) then
        fault.statuscode = Fault_Permission;
fault.write = TRUE;
elseif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((acctype == AccType_IFETCH && (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
(acctype != AccType_IFETCH && walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

Shared Pseudocode Functions
// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32_EL2Enabled(ss) &&
    (if ELStateUsingAAArch32(EL2, ss == SS_Secure) then HCR_VM else HCR_EL2_VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = StageOA(ZeroExtend(va), walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);
return (fault, ipa);
AArch32.S1TranslateSD()

Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor, SDFType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime, SecurityState ss, bits(32) va, AccType acctype, boolean aligned, boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fs1walk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
    return (fault, ipa, SDFType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, ss, va, ispriv);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, SDFType UNKNOWN);

domain = AArch32.OutputDomain(regime, walkstate.domain);
SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

bit ntlsmd;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
    when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
    when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else ntlsmd = '1';

    if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif !acctype IN {AccType_IC, AccType_DC}) && domain == Domain_NoAccess then
        fault.statuscode = Fault_Domain;
    elsif domain == Domain_Client then
        if IsAtomicRW(acctype) then
            if AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, FALSE) then
                fault.statuscode = Fault_Permission;
                fault.write = FALSE;
            elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, TRUE) then
                fault.statuscode = Fault_Permission;
                fault.write = TRUE;
            elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstatepermissions, walkstate.memattrs.memtype, walkstate.baseaddress.paspace, ispriv, acctype, iswrite) then
                fault.statuscode = Fault_Permission;

            if fault.statuscode != Fault_None then
                fault.domain = walkstate.domain;
                return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

            MemoryAttributes memattrs;
            if ((acctype == AccType_IFETCH && (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) || (acctype != AccType_IFETCH &&
walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime)) then
  // Treat memory attributes as Normal Non-Cacheable
  memattrs = NormalNCMemAttr();
  memattrs.xs = walkstate.memattrs.xs;
else
  memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL1 && AArch32.EL2Enabled(ss) &&
  (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
  !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
  memattrs.shareability = walkstate.memattrs.shareability;
else
  memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);
return (fault, ipa, walkstate.sdftype);
Library pseudocode for aarch32/translation/translation/AArch32.S2Translate
// AArch32.S2Translate()
// =====================
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa, SecurityState ss, boolean s2fs1walk, AccType acctype, boolean aligned, boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
assert IsZero(ipa.paddress.address<51:40>);
if !ELStateUsingAArch32(EL2, ss == SS_Secure) then
  s1aarch64 = FALSE;
return AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None;
fault.secondstage = TRUE;
fault.s2fs1walk   = s2fs1walk;
fault.ipaddress   = ipa.paddress;
walkparams = AArch32.GetS2TTWParams();
if walkparams.vm == '0' then
  // Stage 2 is disabled
  return (fault, ipa);
if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
  fault.statuscode = Fault_Translation;
fault.level      = 1;
return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S2Walk(fault, walkparams, ipa);
if fault.statuscode != Fault_None then
  return (fault, AddressDescriptor UNKNOWN);
if AArch32.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
  fault.statuscode = Fault_Alignment;
elsif IsAtomicRW(acctype) then
  assert !s2fs1walk; // AArch32 does not support HW update of TT
  if AArch32.S2HasPermissionsFault(s2fs1walk, walkparams, walkstate.permissions, walkstate.memattrs.memtype, ispriv, acctype, FALSE) then
    // The permission fault was not caused by lack of write permissions
    fault.statuscode = Fault_Permission;
fault.write      = FALSE;
elsif AArch32.S2HasPermissionsFault(s2fs1walk, walkparams, walkstate.permissions, walkstate.memattrs.memtype, ispriv, acctype, TRUE) then
  // The permission fault _was_ caused by lack of write permissions
  fault.statuscode = Fault_Permission;
fault.write      = TRUE;
elsif AArch32.S2HasPermissionsFault(s2fs1walk, walkparams, walkstate.permissions, walkstate.memattrs.memtype, ispriv, acctype, iswrite) then
  fault.statuscode = Fault_Permission;
MemoryAttributes s2_memattrs;
if ((s2fs1walk &&
  walkstate.memattrs.memtype == MemType_Device) ||
  (acctype == AccType_IFETCH &&
  walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
  (acctype != AccType_IFETCH &&
  walkstate.memattrs.memtype == MemType_Normal && HCR2.CD == '1')) then
Treat memory attributes as Normal Non-Cacheable
s2_memattrs = NormalNCMemAttr();
else
s2_memattrs = walkstate.memattrs;
memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.tgx, walkstate);
pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
return (fault, pa);

Library pseudocode for aarch32/translation/translation/AArch32.SDStageOA

// AArch32.SDStageOA()
// ===================
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFType sdftype)
integer tsize;
case sdftype of
when SDFType_SmallPage tsize = 12;
when SDFType_LargePage tsize = 16;
when SDFType_Section tsize = 20;
when SDFType_Supersection tsize = 24;

// Output Address
FullAddress oa;
oa.address = baseaddress.address<51:tsize>:va<tsize-1:0>;
oa.paspace = baseaddress.paspace;
return oa;

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

// AArch32.TranslateAddress()
// =========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccType acctype,
boolean iswrite, boolean aligned,
integer size)
regime = TranslationRegime(PSTATE.EL, acctype);
if !RegimeUsingAArch32(regime) then
return AArch64.TranslateAddress(ZeroExtend(va, 64), acctype, iswrite, aligned, size);
result = AArch32.FullTranslate(va, acctype, iswrite, aligned);
if !IsFault(result) then
result.fault = AArch32.CheckDebug(va, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va);
return result;
// AArch32.DecodeDescriptorTypeLD
// ================================
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
  if descriptor<1:0> == '11' && level == FINAL_LEVEL then
    return DescriptorType_Page;
  elsif descriptor<1:0> == '11' then
    return DescriptorType_Table;
  elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
    return DescriptorType_Block;
  else
    return DescriptorType_Invalid;

// AArch32.DecodeDescriptorTypeSD
// ================================
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
  if level == 1 && descriptor<1:0> == '01' then
    return SDFType_Table;
  elsif level == 1 && descriptor<18,1> == '01' then
    return SDFType_Section;
  elsif level == 1 && descriptor<18,1> == '11' then
    return SDFType_Supersection;
  elsif level == 2 && descriptor<1:0> == '01' then
    return SDFType_LargePage;
  elsif level == 2 && descriptor<1:0> == '1x' then
    return SDFType_SmallPage;
  else
    return SDFType_Invalid;

// AArch32.S1IASize
// ==================
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
  return 32 - UInt(txsz);
Library pseudocode for aarch32/translation/walk/AArch32.S1WalkLD
// AArch32.S1WalkLD()
// ==============
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime, SecurityState ss, S1TTWParams walkparams, bits(32) va, boolean ispriv)

    FaultRecord fault = fault_in;
    bits(3) txsz;
    bits(64) ttbr;
    bit epd;
    if regime == Regime_EL2 then
        ttbr = HTTBR;
        txsz = walkparams.t0sz;
    else
        varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        bits(64) ttbr0;
        bits(64) ttbr1;
        TTBCR_Type ttbcr;
        if regime == Regime_EL30 then
            ttbcr = TTBCR_S;
            ttbr0 = TTBR0_S;
            ttbr1 = TTBR1_S;
        elsif HaveAArch32EL(EL3) then
            ttbcr = TTBCR_NS;
            ttbr0 = TTBR0_NS;
            ttbr1 = TTBR1_NS;
        else
            ttbcr = TTBCR;
            ttbr0 = TTBR0;
            ttbr1 = TTBR1;
        assert ttbcr.EAE == '1';
        if varange == VARange_LOWER then
            txsz = walkparams.t0sz;
        ttbr = ttbr0;
        epd  = ttbcr.EPD0;
        else
            txsz = walkparams.t1sz;
            ttbr = ttbr1;
            epd  = ttbcr.EPD1;
        if regime != Regime_EL2 && epd == '1' then
            fault.level      = 1;
            fault.statuscode = Fault_Translation;
            return (fault, TTWState UNKNOWN);
    // Input Address size
    isize      = AArch32.S1IASize(txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;
    startlevel  = FINAL_LEVEL - (((isize-1) - granulebits) DIV stride);
    levels      = FINAL_LEVEL - startlevel;
    if !IsZero(ttbr<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);
    FullAddress baseaddress;
    baselsb = (isize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb));

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
walkstate.nG = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table = '00';
walkstate.permissions.xn_table = '0';
walkstate.permissions.pxn_table = '0';

indexmsb = isize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;
walkaddress.vaddress = ZeroExtend(va);
if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
    (!boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")
) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

integer indexlsb;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
    bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000');
    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
        s2fs1walk = TRUE;
        s2acctype = AccType_TTW;
        s2aligned = TRUE;
        s2write = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fs1walk, s2acctype, s2aligned, s2write, ispriv);
        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);
    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);
    desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case desctype of
        when DescriptorType_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault_AddressSize;
                return (fault, TTWState UNKNOWN);
        walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12));
        if walkstate.baseaddress.paspace == PAS_Secure && descriptor<63> == '1' then
            walkstate.baseaddress.paspace = PAS_NonSecure;
        if walkparams.hpd == '0' then
            Shared Pseudocode Functions Page 2848
walkstate.permissions.xn_table = (walkstate.permissions.xn_table OR descriptor<60>);
walkstate.permissions.ap_table = (walkstate.permissions.ap_table OR descriptor<62:61>);
walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR descriptor<59>);

walkstate.level = walkstate.level + 1;
indexmsb = indexlsb - 1;

when DescriptorType_Invalid
    fault.statuscode = Fault_Translation;
    return (fault, TTWState UNKNOWN);

when DescriptorType_Page, DescriptorType_Block
    walkstate.istable = FALSE;

until desctype IN {DescriptorType_Page, DescriptorType_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState UNKNOWN);

walkstate.permissions.xn = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap = descriptor<7:6>:1';
walkstate.contiguous = descriptor<52>;
if regime == Regime_EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elsif ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor<11>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));
if walkstate.baseaddress.paspace == PAS_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS_NonSecure;

memattr = descriptor<4:2>;
sh = descriptor<9:8>;
attr = MAIRAttr(UInt(memattr), walkparams.mair);
slaarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64);
return (fault, walkstate);
Library pseudocode for aarch32/translation/walk/AArch32.S1WalkSD
AArch32.S1WalkSD()
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime, SecurityState ss, bits(32) va, boolean ispriv)

FaultRecord fault = fault_in;
SCTLR_Type sctlr;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;

// Determine correct translation control registers to use.
if regime == Regime_EL30 then
    sctlr = SCTLR_S;
    ttbcr = TTBCR_S;
    ttbr0 = TTBR0_S;
    ttbr1 = TTBR1_S;
elsif HaveAArch32EL(EL3) then
    sctlr = SCTLR_NS;
    ttbcr = TTBCR_NS;
    ttbr0 = TTBR0_NS;
    ttbr1 = TTBR1_NS;
else
    sctlr = SCTLR;
    ttbcr = TTBCR;
    ttbr0 = TTBR0;
    ttbr1 = TTBR1;

assert ttbcr.EAE == '0';
ee  = sctlr.EE;
afe = sctlr.AFE;
tre = sctlr.TRE;
n = UInt(ttbcr.N);
bits(32) ttb;
bis(1) pd;
bis(2) irgn;
bis(2) rgn;
bis(1) s;
bis(1) nos;
if n == 0 || IsZero(va<31:(32-n)) then
    ttb  = ttbr0.TTB0:Zeros(7);
    pd   = ttbr0.PD0;
    irgn = ttbr0.IRGN;
    rgn  = ttbr0.RGN;
    s    = ttbr0.S;
    nos  = ttbr0.NOS;
else
    n    = 0;  // TTBR1 translation always treats N as 0
    ttb  = ttbr1.TTB1:Zeros(7);
    pd   = ttbr1.PD1;
    irgn = ttbr1.IRGN;
    rgn  = ttbr1.RGN;
    s    = ttbr1.S;
    nos  = ttbr1.NOS;

// Check if Translation table walk disabled for translations with this Base register.
if pd == '1' then
    fault.level      = 1;
    fault.statuscode = Fault_Translation;
return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n));

TTWState walkstate;
walkstate.baseaddress = baseaddress;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global. Translations in Short-Descriptor Format
// always support global & non-global translations.
walkstate.nG   = '1';
walkstate.memattrs = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level = 1;
walkstate.istable = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
    fault.level = walkstate.level;
    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00');
    else
        index = ZeroExtend(va<19:12>:'00');

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
        s2fs1walk = TRUE;
        s2acctype = AccType_TTW;
        s2aligned = TRUE;
        s2write  = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fs1walk, s2acctype, s2aligned, s2write, ispriv);

        if s2fault.statuscode != Fault_None then
            return (s2fault, TTwState UNKNOWN);
        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, fault);

        if fault.statuscode != Fault_None then
            return (fault, TTwState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDFType_Invalid
            fault.domain     = domain;
when `SDFType_Table`
  domain = descriptor<8:5>;
  ns     = descriptor<3>;
  pxn    = descriptor<2>;

  walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10));
  walkstate.level = 2;

when `SDFType_SmallPage`
  nG  = descriptor<11>;
  s   = descriptor<10>;
  ap  = descriptor<9,5:4>;
  tex = descriptor<8:6>;
  c   = descriptor<3>;
  b   = descriptor<2>;
  xn  = descriptor<0>;

  walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12));
  walkstate.istable = FALSE;

when `SDFType_LargePage`
  xn  = descriptor<15>;
  tex = descriptor<14:12>;
  nG  = descriptor<11>;
  s   = descriptor<10>;
  ap  = descriptor<9,5:4>;
  c   = descriptor<3>;
  b   = descriptor<2>;

  walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16));
  walkstate.istable = FALSE;

when `SDFType_Section`
  ns     = descriptor<19>;
  nG     = descriptor<17>;
  s      = descriptor<16>;
  ap     = descriptor<15,11:10>;
  tex    = descriptor<14:12>;
  domain = descriptor<8:5>;
  xn     = descriptor<4>;
  c      = descriptor<3>;
  b      = descriptor<2>;
  pxn    = descriptor<0>;

  walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20));
  walkstate.istable = FALSE;

when `SDFType_Supersection`
  ns     = descriptor<19>;
  nG     = descriptor<17>;
  s      = descriptor<16>;
  ap     = descriptor<15,11:10>;
  tex    = descriptor<14:12>;
  xn     = descriptor<4>;
  c      = descriptor<3>;
  b      = descriptor<2>;
  pxn    = descriptor<0>;
  domain = '0000';

  walkstate.baseaddress.address = ZeroExtend(descriptor<8,5,23:20,31:24>:Zeros(24));
  walkstate.istable = FALSE;

until walkstate.sdftype != `SDFType_Table`;

if afe == '1' && ap<0> == '0' then
  fault.domain     = domain;
  fault.statuscode = Fault_AccessFlag;
return (fault, TTWState UNKNOWN);

// Decode the TEX, C, B and S bits to produce target memory attributes
if tre == '1' then
    walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
elsif RemapRegsHaveResetValues() then
    walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
else
    walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

walkstate.permissions.ap = ap;
walkstate.permissions.xn = xn;
walkstate.permissions.pxn = pxn;
waklstate.domain = domain;
waklstate.nG = nG;

if ss == SS_Secure && ns == '0' then
    walkstate.baseaddress.paspace = PAS_Secure;
else
    walkstate.baseaddress.paspace = PAS_NonSecure;

return (fault, walkstate);

Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

// AArch32.S2IASize()
// ============
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
return 32 - SInt(t0sz);

Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

// AArch32.S2StartLevel()
// =============
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
return 2 - UInt(sl0);
AArch32.S2Walk()

// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams, AddressDescriptor ipa)

FaultRecord fault = fault_in;

if walkparams.sl0 == '1x' || AArch32.S2InconsistentSL(walkparams) then
    fault.statuscode = Fault_Translation;
    fault.level = 1;
    return (fault, TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S2IASize(walkparams.t0sz);
startlevel = AArch32.S2StartLevel(walkparams.sl0);
levels = FINAL_LEVEL - startlevel;
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;

if !IsZero(VTTBR<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = PAS_NonSecure;
baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb));

TTWState walkstate;
walkstate.baseaddress = baseaddress;
wwalkstate.level = startlevel;
wwalkstate.istable = TRUE;
wwalkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ipa.vaddress;
if HCR2.CD == '1' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

integer indexlsb;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
    bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:000');
    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);
    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);
    desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);
    case desctype of
        when DescriptorType_Table
            // Shared Pseudocode Functions
if !IsZero(descriptor<47:40>) then
  fault.statuscode = Fault_AddressSize;
  return (fault, TTWState UNKNOWN);
walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12));
walkstate.level = walkstate.level + 1;
indexmsb = indexlsb - 1;

when DescriptorType_Invalid
  fault.statuscode = Fault_Translation;
  return (fault, TTWState UNKNOWN);
when DescriptorType_Page, DescriptorType_Block
  walkstate.istable = FALSE;
until desctype IN {DescriptorType_Page, DescriptorType_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
  fault.statuscode = Fault_AddressSize;
  return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
  fault.statuscode = Fault_AccessFlag;
  return (fault, TTWState UNKNOWN);

// Unpack the descriptor into address and upper and lower block attributes
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));

walkstate.permissions.s2ap = descriptor<7:6>;
walkstate.permissions.s2xn = descriptor<54>;
if HaveExtendedExecuteNeverExt() then
  walkstate.permissions.s2xnx = descriptor<53>;
else
  walkstate.permissions.s2xnx = '0';
memattr = descriptor<5:2>;
sh = descriptor<9:8>;
walkstate.memattrs = S2DecodeMemAttrs(memattr, sh);
walkstate.contiguous = descriptor<52>;
return (fault, walkstate);

Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

// AArch32.TranslationSizeSD()
// ===========================
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDType sdftype)
  integer tsize;
  case sdftype of
    when SDType_SmallPage    tsize = 12;
    when SDType_LargePage    tsize = 16;
    when SDType_Section      tsize = 20;
    when SDType_Supersection tsize = 24;
  return tsize;

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

boolean RemapRegsHaveResetValues();
Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

// AArch32.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
S1TTWParams walkparams;

case regime of
  when Regime_EL2         walkparams = AArch32.S1TTWParamsEL2();
  when Regime_EL10        walkparams = AArch32.S1TTWParamsEL10(va);
  when Regime_EL30        walkparams = AArch32.S1TTWParamsEL30(va);

return walkparams;

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

// AArch32.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
S2TTWParams walkparams;

walkparams.tgx = TGx_4KB;
walkparams.s  = VTCR.S;
walkparams.t0sz = VTCR.T0SZ;
walkparams.sl0 = VTCR.SL0;
walkparams.irgn = VTCR.IRGN0;
walkparams.orgn = VTCR.ORGN0;
walkparams.sh  = VTCR.SH0;
walkparams.ee  = HSCTLr.EE;
walkparams.ptw = HCR.PTW;
walkparams.vm  = HCR.VM OR HCR.DC;

// VTCR.S must match VTCR.T0SZ[3]
if walkparams.s != walkparams.t0sz<3> then
  (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS);

return walkparams;

Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

// AArch32.GetVARange()
// ====================
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)

// Lower range Input Address size
lo_iaszize = AArch32.S1IASize(t0sz);
// Upper range Input Address size
up_iaszize = AArch32.S1IASize(t1sz);

if t1sz == '000' & t0sz == '000' then
  return VARangeLOWER;
elsif t1sz == '000' then
  return if IsZero(va<31:lo_iaszize>) then VARangeLOWER else VARange_UPPER;
elsif t0sz == '000' then
  return if IsZero(va<31:up_iaszize>) then VARangeLOWER else VARange_UPPER;
elsif IsZero(va<31:lo_iaszize>) then
  return VARangeLOWER;
elsif IsOnes(va<31:up_iaszize>) then
  return VARange_UPPER;
else
  // Will be reported as a Translation Fault
  return VARange UNKNOWN;
Library pseudocode for aarch32/translation/walkparams/AArch32.S1DCacheEnabled

// AArch32.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
{
    case regime of
    when Regime_EL30 return SCTLR_S.C == '1';
    when Regime_EL2 return HSCTLR.C == '1';
    when Regime_EL10 return (if HaveAAArch32EL(EL3) then SCTLR_NS.C else SCTLR.C) == '1';
}

Library pseudocode for aarch32/translation/walkparams/AArch32.S1ICacheEnabled

// AArch32.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
{
    case regime of
    when Regime_EL30 return SCTLR_S.I == '1';
    when Regime_EL2 return HSCTLR.I == '1';
    when Regime_EL10 return (if HaveAAArch32EL(EL3) then SCTLR_NS.I else SCTLR.I) == '1';
}
// AArch32.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
bits(64) mair;
bit sif;
TTBCR_Type ttbcr;
TTBCR2_Type ttbcr2;
SCTLR_Type sctlr;

if HaveAArch32EL3() then
    ttbcr = TTBCR_NS;
    ttbcr2 = TTBCR2_NS;
    sctlr = SCTLR_NS;
    mair = MAIR1_NS:MAIR0_NS;
    sif = SCR.SIF;
else
    ttbcr = TTBCR;
    ttbcr2 = TTBCR2;
    sctlr = SCTLR;
    mair = MAIR1:MAIR0;
    sif = SCR_EL3.SIF;

assert ttbcr.EAE == '1';
S1TTWParams walkparams;

walkparams.t0sz = ttbcr.T0SZ;
walkparams.t1sz = ttbcr.T1SZ;
walkparams.ee = sctlr.EE;
walkparams.wxn = sctlr.WXN;
walkparams.uwxn = sctlr.UWXN;
walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then sctlr.nTLSMD else '1';
walkparams.mair = mair;
walkparams.sif = sif;

varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
if varange == VARange_LOWER then
    walkparams.sh = ttbcr.SH0;
    walkparams.irgn = ttbcr.IRGN0;
    walkparams.orgn = ttbcr.ORGN0;
    walkparams.hpd = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD0 else '0';
else
    walkparams.sh = ttbcr.SH1;
    walkparams.irgn = ttbcr.IRGN1;
    walkparams.orgn = ttbcr.ORGN1;
    walkparams.hpd = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD1 else '0';

return walkparams;
Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

// AArch32.S1TTWParamsEL2()
// ------------------------
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()

    S1TTWParams walkparams;

    walkparams.tgx = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.orgn = HTCR.ORGN0;
    walkparams.sh   = HTCR.ORGN0;
    walkparams.hpd  = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee   = HSCTRL.EE;
    walkparams.wxn  = HSCTRL.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = HSCTRL.nTLSMD;
    else
        walkparams.ntlsmd = '1';
    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

// AArch32.S1TTWParamsEL30()
// ------------------------
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)

    assert TTBCR_S.EAE == '1';

    S1TTWParams walkparams;

    walkparams.t0sz = TTBCR_S.T0SZ;
    walkparams.t1sz = TTBCR_S.T1SZ;
    walkparams.ee  = SCTLR_S.EE;
    walkparams.wxn = SCTLR_S.WXN;
    walkparams.uwxn = SCTLR_S.UWXN;
    walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR_S.nTLSMD else '1';
    walkparams.mair = MAIR1_S:MAIR0_S;
    walkparams.sif  = SCR.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARRange_LOWER then
        walkparams.sh   = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGN0;
        walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD0 else '0';
    else
        walkparams.sh   = TTBCR_S.SH1;
        walkparams.irgn = TTBCR_S.IRGN1;
        walkparams.orgn = TTBCR_S.ORGN1;
        walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';

    return walkparams;
Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)

assert !ELUsingAArch32(S1TranslationRegime());
assert n < NumBreakpointsImplemented();

enabled = DBGBCR_EL1[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGCR_EL1[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

if HaveAArch32() && size == 4 then                    // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match the address of the
    // second halfword of an instruction, but not the address of the first halfword, it is
    // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
    // event.
    match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
    if !value_match && match_i then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '111' then
    // The above notwithstanding, if DBGBCR_EL1[n].BAS == '111', then it is CONSTRAINED
    // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
    // at the address DBGVR_EL1[n]+2.
    if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

match = value_match && state_match && enabled;
return match;
```

Shared Pseudocode Functions
Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointValueMatch
boolean AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress, boolean linked_to)

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.
integer n = n_in;

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
// no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if n >= NumBreakpointsImplemented() then
    Constraint c;
    (c, n) = ConstraintUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
// call from StateMatch for linking).
if DBGBCR_EL1[n].E == '0' then return FALSE;

context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
// Determine what to compare against.
match_addr = (dbgtype == '001x');
match_vmid = (dbgtype == '10xx');
match_cid  = (dbgtype == '001x');
match_cid1 = (dbgtype IN { '101x', 'x11x'});
match_cid2 = (dbgtype == '11xx');
linked     = (dbgtype == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
// linked_to && !linked && !match_addr then return FALSE;

// Do the comparison.
boolean BVR_match;
if match_addr then
    boolean byte_select_match;
    byte = UInt(vaddress<1:0>);
    if HaveAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
        // If the DBGxVR<=> EL1.RESS field bits are not a sign extension of the MSB
        // of DBGVBR<=> EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
        // generates a Translation fault.
    }
    } else
        assert byte == 0; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    }
    return byte_select_match;
else
    return FALSE;

// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
// Determine what to compare against.
match_addr = (dbgtype == '001x');
match_vmid = (dbgtype == '10xx');
match_cid  = (dbgtype == '001x');
match_cid1 = (dbgtype IN { '101x', 'x11x'});
match_cid2 = (dbgtype == '11xx');
linked     = (dbgtype == 'xxx1');

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, of if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
// linked_to && !linked && !match_addr then return FALSE;

// Do the comparison.
boolean BVR_match;
if match_addr then
    boolean byte_select_match;
    byte = UInt(vaddress<1:0>);
    if HaveAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2}; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0; // "vaddress" is word aligned
        byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is RES1
        // If the DBGxVR<=> EL1.RESS field bits are not a sign extension of the MSB
        // of DBGVBR<=> EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
        // generates a Translation fault.
    }
    } else
        assert byte == 0; // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    }
    return byte_select_match;
else
    return FALSE;
integer top = AArch64.VAMax();
if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
  if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
    top = 63;
  BVR_match = (vaddress<top:2> == DBGBVR_EL1[n]<top:2>) && byte_select_match;
elsif match_cid then
  if IsInHost() then
    BVR_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
  else
    BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
elsif match_cid1 then
  BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
boolean BXVR_match;
if match_vmid then
  bits(16) vmid;
  bits(16) bvr_vmid;
  if !Have16bitVMID() || VTCR_EL2.VS == '0' then
    vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
  else
    vmid = VTTBR_EL2.VMID;
    bvr_vmid = DBGBVR_EL1[n]<47:32>;
  BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
    !IsInHost() &&
    vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug()) &&
    EL2Enabled() &&
    DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);
match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return match;
// AArch64.StateMatch()
// =============
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC_in, bit HMC_in, bits(2) PxC_in, boolean linked_in, bits(4) LBN, boolean isbreakpnt, AccType accype, boolean ispriv)

// "SSC_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

bits(2) SSC = SSC_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
if c == Constraint_DISABLED then return FALSE;

EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

boolean priv_match;
if HaveNV2Ext() && accype == AccType_NV2REGISTER && !isbreakpnt then
  priv_match = EL2_match;
elsif !ispriv && !isbreakpnt then
  priv_match = EL0_match;
else
  case PSTATE.EL of
    when EL3 priv_match = EL3_match;
    when EL2 priv_match = EL2_match;
    when EL1 priv_match = EL1_match;
    when EL0 priv_match = EL0_match;
  endcase

boolean security_state_match;
ss = CurrentSecurityState();

case security_state_match of
  when '00' security_state_match = TRUE;                     // Both
  when '01' security_state_match = ss == SS_NonSecure;       // Non-secure only
  when '10' security_state_match = ss == SS_Secure;         // Secure only
  when '11' security_state_match = (HMC == '1' || ss == SS_Secure); // HMC=1 -> Both, 0 -> Secure
endcase

integer lbn;
if linked then
  // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
  // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
  // UNKNOWN breakpoint that is context-aware.
  lbn = UInt(LBN);
  first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
  last_ctx_cmp = NumBreakpointsImplemented() - 1;
  if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
    (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
  endcase
  assert c IN (Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN);

boolean linked_match;
if linked then
  vaddress = bits(64) UNKNOWN;
  linked_to = TRUE;
  linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

Shared Pseudocode Functions
return priv_match && security_state_match && (!linked || linked_match);

**Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions**

```java
// AArch64.GenerateDebugExceptions()
// --------------------------------
boolean AArch64.GenerateDebugExceptions()
return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

**Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom**

```java
// AArch64.GenerateDebugExceptionsFrom()
// -------------------------------------
boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)
if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
    return FALSE;

route_to_el2 = HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if from == EL0 && ELUsingAArch32(EL1) then
        enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);
    return enabled;
```

**Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow**

```java
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events
AArch64.CheckForPMUOverflow()
boolean pmuirq;
bit E;
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1.C == '1' && PMOVSET_EL0.C == '1';
target = GetNumEventCounters();
if counters != 0 then
    for idx = 0 to counters - 1
        if pmuirq then
            SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
            CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
// The request remains set until the condition is cleared. (For example, an interrupt handler
// or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLEL0.)
```
// AArch64.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch64.ClearEventCounters()
   integer counters = AArch64.GetNumEventCountersAccessible();
   if counters != 0 then
      for idx = 0 to counters - 1
         PMEVCNTR_EL0[idx] = Zeros();
Library pseudocode for aarch64/debug/pmu/AArch64.CountPMUEvents

Shared Pseudocode Functions
boolean AArch64.CountPMUEvents(integer idx)
assert idx == CYCLE_COUNTER_ID || idx < GetNumEventCounters();
boolean debug;
boolean enabled;
boolean prohibited;
boolean filtered;
boolean frozen;
boolean resvd_for_el2;
bis(32) ovflws;
// Event counting is disabled in Debug state
debug = Halted();
// Software can reserve some counters for EL2
resvd_for_el2 = AArch64.PMUCounterIsHyp(idx);
// Main enable controls
if idx == CYCLE_COUNTER_ID then
    enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
else
    E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';
// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;
// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * MDCR_EL3.SPME == 0, and either:
//   - FEAT_PMUv3p7 is not implemented
//   - MDCR_EL3.MPMX == 0
if HaveEL3() && IsSecure() then
    if HavePMUv3p7() then
        prohibited = MDCR_EL3.<SPME,MPMX> == '00';
    else
        prohibited = MDCR_EL3.SPME == '0';
// Event counting at EL3 is prohibited if all of:
// * FEAT_PMUv3p7 is implemented
// * One of the following is true:
//   - MDCR_EL3.SPME == 0
//   - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
    prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);
// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * MDCR_EL2.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = MDCR_EL2.HPMD == '1';
// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();
// Event counting might be frozen
frozen = FALSE;
// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() then
    ovflws = ZeroExtend(PMOVSSET_EL0<GetNumEventCounters()-1:0>);
    if resvd_for_el2 then
        FZ = MDCR_EL2.HPMFZO;
ovflws<UInt(MDCR_EL2.HPMN)-1:0> = Zeros();
else
    FZ = PMCR_EL0.FZ0;
    if HaveEL(EL2) && UInt(MDCR_EL2.HPMN) < GetNumEventCounters() then
        ovflws<GetNumEventCounters()-1:UIInt(MDCR_EL2.HPMN)> = Zeros();

    frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR_EL0.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    enabled = enabled && (PMCR_EL0.DP == '0');
// Otherwise whether event counting is prohibited does not affect the cycle counter
    prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && IsSecure() && MDCR_EL3.SCCD == '1' then
        prohibited = TRUE;
    if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
        prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
// This is not overriden by PMCR_EL0.DP.
if HavePMUv3p7() && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
    filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR_EL0<31:0> else PMEVTYPER_EL0[idx]<31:0>;

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M = if HaveEL(EL3) then filter<26> else '0';
    SH = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';

    ss = CurrentSecurityState();
    case PSTATE.EL of
        when EL0 filtered = if ss == SS_Secure then U else U != NSU;
        when EL1 filtered = if ss == SS_Secure then P != NSK;
        when EL2 filtered = if ss == SS_Secure then NSH != SH else NSH == '0';
        when EL3 filtered = if M != P;
    return !debug && enabled && !prohibited && !filtered && !frozen;

Library pseudocode for aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

// AArch64.GetNumEventCountersAccessible()
//  =======================================
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
integer n;
integer total_counters = GetNumEventCounters();
// Software can reserve some counters for EL2
if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
    n = UInt(MDCR_EL2.HPMN);
    if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
        n = ConstrainUnpredictableInteger(0, total_counters, Unpredictable_PMUEVENTCOUNTER);
else
    n = total_counters;
return n;

Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```c
// AArch64.IncrementEventCounter()
// ===============================================
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);
    if HavePMUv3p5() then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        lp = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
        ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>);
        ovflw = 32;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
        PMOVSCLR_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() && (!HavePMUv3p5() || lp == '0') then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUCounterIsHyp

```c
// AArch64.PMUCounterIsHyp
// =======================
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch64.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
        resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN);
    if UInt(MDCR_EL2.HPMN) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(MDCR_EL2.HPMN)) then
        resvd_for_el2 = boolean UNKNOWN;
    else
        resvd_for_el2 = FALSE;
    return resvd_for_el2;
```

Shared Pseudocode Functions
Library pseudocode for aarch64/debug/pmui/AArch64.PMUCycle

// AArch64.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
if !HavePMUv3() then
    return;
PMUEvent(PMU_EVENT_CPU_CYCLES);

integer counters = GetNumEventCounters();
if counters != 0 then
    for idx = 0 to counters - 1
        if AArch64.CountPMUEvents(idx) then
            accumulated = PMUEventAccumulator[idx];
            AArch64.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;

integer old_value;
infrared new_value;
infrared ovflw;
if (AArch64.CountPMUEvents(CYCLE_COUNTER_ID) &&
    (!HaveAArch32() || PMCR_EL0.LC == '1' || PMCR_EL0.D == '0' || Has.Elapsed64Cycles()) then
    old_value = UInt(PMCCNTR_EL0);
    new_value = old_value + 1;
    PMCCNTR_EL0 = new_value<63:0>;
    if HaveAArch32() then
        ovflw = if PMCR_EL0.LC == '1' then 64 else 32;
    else
        ovflw = 64;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0.C = '1';
        PMOVSCLR_EL0.C = '1';
AArch64.CheckForPMUOverflow();

Library pseudocode for aarch64/debug/pmui/AArch64.PMUSwIncrement

// AArch64.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMSWINC_EL0.

AArch64.PMUSwIncrement(bits(32) sw_incr)
integer counters = AArch64.GetNumEventCountersAccessible();
if counters != 0 then
    for idx = 0 to counters - 1
        if sw_incr<idx> == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
if !StatisticalProfilingEnabled() then return FALSE;
if PSTATE.EL == EL2 then return FALSE;
if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
return PMSCR_EL1.CX == '1';
Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```java
// CollectContextIDR2()
// ===============

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```java
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```
// CollectTimeStamp()
// ===============

TimeStamp CollectTimeStamp()
{
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (., owning_el) = ProfilingBufferOwner();
    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
        else
            if PMSCR_EL1.TS == '0' then return TimeStamp_None;
            bits(2) PCT_el1;
            if !HaveECVExt() then
                PCT_el1 = '0':PMSCR_EL1.PCT<0>;    // PCT<1> is RES0
            else
                PCT_el1 = PMSCR_EL1.PCT;
                if PCT_el1 == '10' then
                    // Reserved value
                    (., PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
            
            if EL2Enabled() then
                bits(2) PCT_el2;
                if !HaveECVExt() then
                    PCT_el2 = '0':PMSCR_EL2.PCT<0>;    // PCT<1> is RES0
                else
                    PCT_el2 = PMSCR_EL2.PCT;
                    if PCT_el2 == '10' then
                        // Reserved value
                        (., PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
                
                case PCT_el2 of
                    when '00' return TimeStamp_Virtual;
                    when '01'
                        if owning_el == EL2 then return TimeStamp_Physical;
                        when '11'
                            assert HaveECVExt();    // FEAT_ECV must be implemented
                            if owning_el == EL1 && PCT_el1 == '00' then
                                return TimeStamp_Virtual;
                            else
                                return TimeStamp_OffsetPhysical;
                    otherwise
                        Unreachable();
            
            case PCT_el1 of
                when '00' return TimeStamp_Virtual;
                when '01' return TimeStamp_Physical;
                when '11'
                    assert HaveECVExt();    // FEAT_ECV must be implemented
                    return TimeStamp_OffsetPhysical;
                otherwise
                    Unreachable();
    }
    
    enumeration OpType {
        OpType_Load,    // Any memory-read operation other than atomics, compare-and-swap, and swap
        OpType_Store,   // Any memory-write operation, including atomics without return
        OpType_LoadAtomic,   // Atomics with return, compare-and-swap and swap
        OpType_Branch,    // Software write to the PC
        OpType.Other      // Any other class of operation
    }
Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
// ProfilingBufferEnabled()
// ========================

if !HaveStatisticalProfiling() then return FALSE;
(owning_ss, owning_el) = ProfilingBufferOwner();
state_match = ((owning_ss == SS_Secure && SCR_EL3.NS == '0') ||
(owning_ss == SS_NonSecure && SCR_EL3.NS == '1'));
return (!ELUsingAArch32(owning_el) && state_match &&
PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

// ProfilingBufferOwner()
// ======================

(SecurityState, bits(2)) ProfilingBufferOwner()
// ProfilingBufferOwner()
// ======================

if HaveEL(EL3) then
  owning_ss = if MDCR_EL3.NSPB<1> == '0' then SS_Secure else SS_NonSecure;
else
  owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;

bits(2) owning_el;
if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
  owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
else
  owning_el = EL1;

return (owning_ss, owning_el);

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

// ProfilingSynchronizationBarrier()
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
// SPECollectRecord()
// ============
// Returns TRUE if the sampled class of instructions or operations, as
determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType optype) {
    // StatisticalProfilingEnabled();

    bits(64) mask = 0xAA<63:0>;                // Bits [7,5,3,1]
    if HaveSVE() then mask<18:17> = Ones();   // Predicate flags
    if HaveStatisticalProfilingv1p1() then mask<11> = '1'; // Alignment Flag
    if HaveStatisticalProfilingv1p2() then mask<6> = '1';   // Not taken flag

    mask<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE mask 63:48";
    mask<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE mask 31:24";
    mask<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE mask 15:12";

    // Check for UNPREDICTABLE case
    if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.<FnE,FE> == '11' && !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
        if ConstrainUnpredictableBool(Unpredictable_BADPMSFCR) then
            return FALSE;
        else
            // Filtering by event
            if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
                e = events AND mask;
                m = PMSEVFR_EL1 AND mask;
                if !IsZero(NOT(e) AND m) then return FALSE;
            // Filtering by inverse event
            if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.FnE == '1' && !IsZero(PMSNEVFR_EL1)) then
                e = events AND mask;
                m = PMSNEVFR_EL1 AND mask;
                if !IsZero(e AND m) then return FALSE;
            // Filtering by type
            if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
                case optype of
                    when OpType_Branch
                        if PMSFCR_EL1.B == '0' then return FALSE;
                    when OpType_Load
                        if PMSFCR_EL1.LD == '0' then return FALSE;
                    when OpType_Store
                        if PMSFCR_EL1.ST == '0' then return FALSE;
                    when OpType_LoadAtomic
                        if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
                    otherwise
                        return FALSE;
            // Filtering by latency
            if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
                if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
                    return FALSE;
            // Check for UNPREDICTABLE cases
            if ((PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1 AND mask)) ||
                (PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
                (PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT))) then
                return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
            if (HaveStatisticalProfilingv1p2() && ((PMSFCR_EL1.FN == '1' && !IsZero(PMSNEVFR_EL1 AND mask)) ||
                (PMSFCR_EL1.<FnE,F> == '1l' && !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)))) then
                return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
            return TRUE;
    return TRUE;
}
// StatisticalProfilingEnabled
// =============================

boolean StatisticalProfilingEnabled()
{
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) < UInt(PSTATE.EL) || (tge_set && owning_el == EL1) ||
        owning_ss != CurrentSecurityState()) then
        return FALSE;

    bit spe_bit;
    case PSTATE.EL of
        when EL3 Unreachable();
        when EL2 spe_bit = PMSCR_EL2.E2SPE;
        when EL1 spe_bit = PMSCR_EL1.E1SPE;
        when EL0 spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);
    return spe_bit == '1';

enum TimeStamp {
    TimeStamp_None, // No timestamp
    TimeStamp_CoreSight, // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical, // Physical counter value with no offset
    TimeStamp_OffsetPhysical, // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual // Physical counter value minus CNTVOFF_EL2
};
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
assert HaveEL(target_el) & !ELUsingAArch32(target_el) & UInt(target_el) >= UInt(PSTATE.EL);
ExceptionRecord exception = exception_in;
boolean sync_errors;
if HaveIESB() then
  sync_errors = SCTLR[target_el].IESB == '1';
if HaveDoubleFaultExt() then
  sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '1' & target_el == EL3);
// SCTLR[].IESB and/or SCR EL3.NMEA (if applicable) might be ignored in Debug state.
if !ConstrainUnpredictableBool(UnpredictableIESBinDebug) then
  sync_errors = FALSE;
else
  sync_errors = FALSE;
SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
MaybeZeroSVEUppers(target_el);
AArch64.ReportException(exception, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = bits(64) UNKNOWN;
ELR[] = bits(64) UNKNOWN;

// PSTATE.<SS,D,A,I,F> are not observable and ignored in Debug state, so behave as if UNKNOWN.
PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
  PSTATE.IT = '00000000';
PSTATE.T = '0'; // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0)))) &
  SCTLR[].SPAN == '0') then
  PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
if HaveMTEExt() then PSTATE.TCO = '1';

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;
EDSCR.ERR = '1';
UpdateEDSCRFields(); // Update EDSCR processor state flags.

if sync_errors then
  SynchronizeErrors();
EndOfInstruction();
// AArch64.WatchpointByteMatch()
// =======================================

boolean AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

    integer top = AArch64.VAMax();
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;  // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASECONTIGUOUS);
            bottom = 3;                        // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE;       // Disabled
            when Constraint_NONE mask = 0;               // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR_match;
    if mask > bottom then
        // If the DBGxVR<n> EL1.RESS field bits are not a sign extension of the MSB
        // of DBGxVR<n> EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        if !IsOnes(DBGGBVR_EL1[n]<63:top>) && !IsZero(DBGGBVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
            WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMAKEDBITS);
        else
            WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

// AArch64.WatchpointMatch()
// ===============
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32($1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    // "ispriv" is:
    //  * FALSE for all loads, stores, and atomic operations executed at EL0.
    //  * FALSE if the access is unprivileged.
    //  * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);

    ls_match = FALSE;
    if acctype == AccType_ATOMICRW then
        ls_match = (DBGWCR_EL1[n].LSC != '00');
    else
        ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

    return value_match && state_match && ls_match && enabled;

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)
    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
            else
                AArch64.WatchpointException(vaddress, fault);
        else
            AArch64.InstructionAbort(vaddress, fault);
        else
            AArch64.DataAbort(vaddress, fault);
Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
exception = ExceptionSyndrome(exctype);

d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_NV2Watchpoint};

(exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);

if IPAValid(fault) then
    exception.ipavalid = TRUE;
else
    exception.ipavalid = FALSE;

return exception;
```
Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) AArch64.EffectiveTCF(AccType acctype)
    bits(2) tcf, el;
    el = S1TranslationRegime();
    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL2.TCF0;
        else
            tcf = SCTLR_EL2.TCF;
        endif
    elsif el == EL1 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL1.TCF0;
        else
            tcf = SCTLR_EL1.TCF;
        endif
    endif
    if tcf == '11' then //reserved value
        if !HaveMTE3Ext() then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF);
        endif
    endif
    return tcf;

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

// AArch64.InstructionAbort()
// ========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
      (HCR_EL2.TGE == '1' ||
      (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
      IsSecondStage(fault)));

    ExceptionRecord exception;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3)) &&
      IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    endif
    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
    bits(2) target_el = EL1;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
// AArch64.PCAlignmentFault()
// =========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr();

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

// AArch64.RaiseTagCheckFault()
// ============================
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, boolean write)

    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_DataAbort);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
        exception.vaddress = bits(4) UNKNOWN : va<59:0>;

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
// AArch64.ReportTagCheckFault()
// =============================
// Records a tag check fault exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
   if el == EL3 then
      assert ttbr == '0';
      TFSR_EL3.TF0 = '1';
   elsif el == EL2 then
      if ttbr == '0' then
         TFSR_EL2.TF0 = '1';
      else
         TFSR_EL2.TF1 = '1';
      end
   elsif el == EL1 then
      if ttbr == '0' then
         TFSR_EL1.TF0 = '1';
      else
         TFSR_EL1.TF1 = '1';
      end
   elsif el == EL0 then
      if ttbr == '0' then
         TFSRE0_EL1.TF0 = '1';
      else
         TFSRE0_EL1.TF1 = '1';
   end

// AArch64.SPAlignmentFault()
// =========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()
   bits(64) preferred_exception_return = ThisInstrAddr();
   vect_offset = 0x0;

   exception = ExceptionSyndrome(Exception_SPAlignment);
   bits(2) target_el = EL1;
   if UInt(PSTATE.EL) > UInt(EL1) then
      target_el = PSTATE.EL;
   elsif EL2Enabled() && HCR_EL2.TGE == '1' then
      target_el = EL2;
   AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
// AArch64.TagCheckFault()  
// =======================  
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccType acctype, boolean iswrite)  
bits(2) tcf, el;  
el = AArch64.AccessUsesEL(acctype);  
tcf = AArch64.EffectiveTCF(acctype);  

```
case tcf of  
    when '00'       // Tag Check Faults have no effect on the PE  
        return;  
    when '01'       // Tag Check Faults cause a synchronous exception  
        AArch64.RaiseTagCheckFault(vaddress, iswrite);  
    when '10'       // Tag Check Faults are asynchronously accumulated  
        AArch64.ReportTagCheckFault(el, vaddress<55>);  
    when '11'       // Tag Check Faults cause a synchronous exception on reads or on  
        // a read-write access, and are asynchronously accumulated on writes  
        // Check for access performing both a read and a write.  
        readwrite = acctype IN { AccType_ATOMICRW,  
                                AccType_ORDERED_ATOMICRW,  
                                AccType_ORDEREDRW };  
        if !iswrite || readwrite then  
            AArch64.RaiseTagCheckFault(vaddress, iswrite);  
        else  
            AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);  
```

// BranchTargetException()  
// =======================  
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)  
bits(64) preferred_exception_return = ThisInstrAddr();  

```
bits(64) preferred_exception_return = ThisInstrAddr();  
vect_offset = 0x0;  

exception = ExceptionSyndrome(Exception_BranchTarget);  
exception.syndrome<1:0> = PSTATE.BTYPE;  
exception.syndrome<24:2> = Zeros();  

bits(2) target_el = EL1;  
if UInt(PSTATE.EL) > UInt(EL1) then  
    target_el = PSTATE.EL;  
elsif PSTATE.EL == EL0 & EL2Enabled() & HCR_EL2.TGE == '1' then  
    target_el = EL2;  
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);  
```
Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalFIQException

// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
              (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1');
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);

if route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
  assert PSTATE.EL != EL3;
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  assert PSTATE.EL IN {EL0, EL1};
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalIRQException

// AArch64.TakePhysicalIRQException()
// ==================================

// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
              (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1');
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);

if route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
  assert PSTATE.EL != EL3;
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  assert PSTATE.EL IN {EL0, EL1};
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

// AArch64.TakePhysicalSErrorException()
// ------------------------------------
AArch64.TakePhysicalSErrorException(bits(25) syndrome)

route_to_el3 = HaveEL(EL3) && SCR_EL3.AA == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
            (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMI == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;

bits(2) target_el;
if PSTATE.EL == EL3 || route_to_el3 then
    target_el = EL3;
elsif PSTATE.EL == EL2 || route_to_el2 then
    target_el = EL2;
else
    target_el = EL1;

if IsSSSErrorEdgeTriggered(target_el, syndrome) then
    ClearPendingPhysicalSError();

exception = ExceptionSyndrome(ExceptionSError);
exception.syndrome = syndrome;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualFIQException

// AArch64.TakeVirtualFIQException()
// ----------------------------------
AArch64.TakeVirtualFIQException()

assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;

exception = ExceptionSyndrome(Exception_IRQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualIRQException

// AArch64.TakeVirtualIRQException()
// ----------------------------------
AArch64.TakeVirtualIRQException()

assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

// AArch64.TakeVirtualSErrorException()
// -----------------------------------

AArch64.TakeVirtualSErrorException(bits(25) syndrome)

assert PSTATE.EL IN {EL0, EL1} & EL2Enabled();
assert HCR_EL2.TGE == '0' & HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;
exception = ExceptionSyndrome(ExceptionSError);

if HaveRASExt() then
  exception.syndrome<24> = VSESR_EL2.IDS;
  exception.syndrome<23:0> = VSESR_EL2.ISS;
else
  impdef_syndrome = syndrome<24> == '1';
  if impdef_syndrome then
    exception.syndrome = syndrome;

ClearPendingVirtualSError();
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

// AArch64.BreakpointException()
// ----------------------------

AArch64.BreakpointException(FaultRecord fault)

assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} & EL2Enabled() & HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

if PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

// AArch64.SoftwareBreakpoint()
// ---------------------------

AArch64.SoftwareBreakpoint(bits(16) immediate)

route_to_el2 = (PSTATE.EL IN {EL0, EL1} & EL2Enabled() & HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);

assert UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarm64/exceptions/debug/AArch64.SoftwareStepException

// AArch64.SoftwareStepException()
// -----------------------------------------

AArch64.SoftwareStepException()
assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
(HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareStep);
if SoftwareStep_DidNotStep() then
  exception.syndrome<24> = '0';
else
  exception.syndrome<24> = '1';
  exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
exception.syndrome<5:0> = '100010'; // IFSC = Debug Exception

if PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarm64/exceptions/debug/AArch64.VectorCatchException

// AArch64.VectorCatchException()
// ----------------------------------------

AArch64.VectorCatchException(FaultRecord fault)
assert PSTATE.EL != EL2;
assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);
AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
// AArch64.WatchpointException()  
// =============================  

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)  
    assert PSTATE.EL != EL3;  
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&  
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));  
    bits(64) preferred_exception_return = ThisInstrAddr();  
    vect_offset = 0x0;  
    ExceptionRecord exception;  
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then  
        exception = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, vaddress);  
    else  
        exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);  
    if PSTATE.EL == EL2 || route_to_el2 then  
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);  
    else  
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer, bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception_Uncategorized           ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap                  ec = 0x01;
        when Exception_CP15RTTrap               ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap              ec = 0x04; assert from_32;
        when Exception_CP14RTTrap               ec = 0x05; assert from_32;
        when Exception_CP14DTTrap               ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap      ec = 0x07;
        when Exception_FPIDTrap                 ec = 0x08;
        when Exception_PACTrap                  ec = 0x09;
        when Exception_UsingAArch32             ec = 0x0A;
        when Exception_CP14RRTTrap              ec = 0x0C; assert from_32;
        when Exception_BranchTarget            ec = 0x0D;
        when Exception_IllegalState             ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall           ec = 0x11;
        when Exception_HypervisorCall           ec = 0x12;
        when Exception_MonitorCall             ec = 0x13;
        when Exception_SystemRegisterTrap       ec = 0x18; assert !from_32;
        when Exception_SVEAccessTrap            ec = 0x19; assert !from_32;
        when Exception_ERetTrap                 ec = 0x1A; assert !from_32;
        when Exception_PACFail                  ec = 0x1C; assert !from_32;
        when Exception_InstructionAbort         ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAppearance             ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort                ec = 0x24;
        when Exception_NV2DataAbort             ec = 0x25;
        when Exception_SPAlignment              ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_MemCpyMemSet             ec = 0x27;
        when Exception_FPTrappedException       ec = 0x28;
        when Exception_SError                   ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint              ec = 0x30; il_is_valid = FALSE;
        when Exception_SoftwareStep             ec = 0x32; il_is_valid = FALSE;
        when Exception_Watchpoint               ec = 0x34; il_is_valid = FALSE;
        when Exception_NV2Watchpoint            ec = 0x35; il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint       ec = 0x38;
        when Exception_VectorCatch              ec = 0x3A; il_is_valid = FALSE; assert from_32;
        otherwise                              Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1';  // AArch64 instructions always 32-bit
    return (ec, il);
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

// exctype = exception.exctype;

(ec, il) = AArch64.ExceptionClass(exctype, target_el);
iss = exception.syndrome;
iss2 = exception.syndrome2;

if ec IN {0x24,0x25} && iss<24> == '0' then
  il = '1';

ESR[target_el] = (Zeros(27) : // <63:37>
  iss2 : // <36:32>
  ec<5:0> : // <31:26>
  il : // <25>
  iss); // <24:0>

if exctype IN {
  Exception_InstructionAbort,
  Exception_PCAIgignment,
  Exception_DataAbort,
  Exception_NV2DataAbort,
  Exception_NV2Watchpoint,
  Exception_Watchpoint
} then
  FAR[target_el] = exception.vaddress;
else
  FAR[target_el] = bits(64) UNKNOWN;

if exception.ipavalid then
  HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
  if IsSecureEL2Enabled() && IsSecure() then
    HPFAR_EL2.NS = exception.NS;
  else
    HPFAR_EL2.NS = '0';
elsif target_el == EL2 then
  HPFAR_EL2<43:4> = bits(40) UNKNOWN;

return;

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch64.ResetControlRegisters(boolean coldReset);
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';  // Select stack pointer
    PSTATE.<D,A,I,F> = '1111';  // All asynchronous exceptions masked
    PSTATE.SS = '0';  // Clear software step bit
    PSTATE.DIT = '0';  // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';  // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;  // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:AArch64.PAMax()>) & IsZero(rv<1:0>);

    boolean branch Conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch Conditional);
Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

// AArch64.FPTrappedException()
// --------------------------------

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
   exception = ExceptionSyndrome(Exception_FPTrappedException);
   if is_ase then
      if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
         exception.syndrome<23> = '1';                          // TFV
      else
         exception.syndrome<23> = '0';                          // TFV
   else
      exception.syndrome<23> = '1';                              // TFV
   exception.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR
   if exception.syndrome<23> == '1' then
      exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
   else
      exception.syndrome<7,4:0> = bits(6) UNKNOWN;
   route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';
   bits(64) preferred_exception_return = ThisInstrAddr();
   vect_offset = 0x0;
   if UInt(PSTATE.EL) > UInt(EL1) then
      AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
   elsif route_to_el2 then
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
   else
      AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

// AArch64.CallHypervisor()
// --------------------------------

AArch64.CallHypervisor(bits(16) immediate)
   assert HaveEL(EL2);
   if UsingAAArch32() then AArch32.ITAdvance();
   SSAAdvance();
   bits(64) preferred_exception_return = NextInstrAddr();
   vect_offset = 0x0;
   exception = ExceptionSyndrome(Exception_HypervisorCall);
   exception.syndrome<15:0> = immediate;
   if PSTATE.EL == EL3 then
      AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
   else
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in, bits(64) preferred_exception_return, integer vect_offset_in)

assert HaveEL(target_el) & !ELUsingAAArch32(target_el) & UInt(target_el) >= UInt(PSTATE.EL);

ExceptionRecord exception = exception_in;

boolean sync_errors;
if HaveIESB() then
    sync_errors = SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    else
        sync_errors = FALSE;

    SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAAArch32();
if from_32 then
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);

integer vect_offset = vect_offset_in;
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAAArch32(EL2);
        else
            lower_32 = ELUsingAAArch32(EL1);
    else
        IsInHost() & PSTATE.EL == EL0 & target_el == EL2 then
            lower_32 = ELUsingAAArch32(EL0);
        else
            lower_32 = ELUsingAAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);
    else if PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState);

    if PSTATE.EL == EL1 & target_el == EL1 & EL2Enabled() then
        if HaveNV2Ext() & (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
            spsr<3:2> = '10';
        else
            if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
                spsr<3:2> = '10';
            if HaveBTIExt() & !UsingAAArch32() then
                boolean zero_btype;
                // SPSR[].BTYPE is only guaranteed valid for these exception types
                if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ, Exception_SoftwareStep, Exception_PCAlignment, Exception_InstructionAbort, Exception_Breakpoint, Exception_VectorCatch, Exception_SoftwareBreakpoint, Exception_IllegalState, Exception_BranchTarget} then
                    zero_btype = FALSE;
                else
                    zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
                    if zero_btype then spsr<11:10> = '00';
                if HaveNV2Ext() & exception.exceptype == Exception_NV2DataAbort & target_el == EL3 then
                    // External aborts are configured to be taken to EL3
                    exception.exceptype = Exception_DataAbort;
if !(exception.exectype IN {Exception_IRQ, Exception_FIQ}) then
            AArch64.ReportException(exception, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT
SCTLR[].SPINTMASK;
PSTATE.<D,A,I,F> = '1111';
PSTATE.I = '0';
if from_32 then                             // Coming from AArch32
            PSTATE.IT = '00000000';
PSTATE.T = '0';                         // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 &&
            ELIsInHost(EL0))) &&
SCTLR[].SPAN == '0') then
            PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
if HaveMTEExt() then PSTATE.TCO = '1';

boolean branch_conditional = FALSE;
BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);

CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();

---

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

// AArch64.AArch32SystemAccessTrap()
// _________________________________
// Trapped AARCH32 system register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
            assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;
exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
AArch64.AArch32SystemAccessTrapSyndrome()

Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)

    ExceptionRecord exception;

    case ec of
    when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
    when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
    when 0x4    exception = ExceptionSyndrome(Exception_CP14RTTrap);
    when 0x5    exception = ExceptionSyndrome(Exception_CP14DTTrap);
    when 0x6    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
    when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
    when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
    when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
    otherwise    Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype == Exception_Uncategorized then return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>;    // opc2
            iss<16:14> = instr<23:21>;   // opc1
            iss<13:10> = instr<19:16>;   // CRn
            iss<4:1>   = instr<3:0>;     // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;   // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MRC, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        endif
    endif

    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        if instr<19:16> == '1111' then    // Rt==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;
        endif

        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
            iss<4:1>   = instr<3:0>;     // CRm
        endif

    elsif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;        // imm8
        iss<4:0>   = instr<23:21>;      // U
        iss<2:1>   = instr<24,21>;      // P,W
        if instr<19:16> == '1111' then  // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = '1';
        endif
    endif

    iss<0> = instr<20>;               // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;
    return exception;
Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```c
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');
    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```c
// AArch64.CheckCP15InstrCoarseTraps()
// ==================================
// Check for coarse-grained AArch32 traps to System registers in the
coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
      (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
      (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
    if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&
        !ELUsingAArch32(EL1) && SCTLR_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);
    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&
            HCR_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        major = if nreg == 1 then CRn else CRm;
        // Check for MRC, MCR, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
        // and MRC and MCR disabled by HCR_EL2.TIDCP.
        if (!IsInHost() && (!major IN {4,14}) && HSTR_EL2<major> == '1') ||
            (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```c
// AArch64.CheckFPAdvSIMDEnabled()
// =================================

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled();
```
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
  // Check if access disabled in CPTR_EL2
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
    boolean disabled;
    case CPTR_EL2.FPEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
      when '11' disabled = FALSE;
      if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
    else
      if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
    endif
  endif
else
  if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
endif

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEnabled

// AArch64.CheckFPEnabled()
// ========================
// Check against CPACR[]

AArch64.CheckFPEnabled()
if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
  // Check if access disabled in CPACR_EL1
  boolean disabled;
  case CPACR_EL1.FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
  endif
else
  if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
endif

AArch64.CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETA, ERETAB instruction
AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
    // is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
                  (HaveNVExt() && HCR_EL2.NV == '1') ||
                  (HaveFGTExt() && HCR_EL2.<E2H, TGE> != '11' &&
                   (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ERET == '1'));

    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then  // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0';  // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then  // ERETA
                exception.syndrome<0> = '0';
            else  // ERETAB
                exception.syndrome<0> = '1';
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction
AArch64.CheckForSMCUndefOrTrap(bits(16) imm)

    if PSTATE.EL == EL0 then UNDEFINED;
    if !(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        exception.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

// AArch64.CheckForSVCTrap()
// ================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL0) && !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' && (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL2) || SCR_EL3.FGTEn == '1')));
        elsif PSTATE.EL == EL1 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HCR_EL2.SVC_EL0 == '1' && (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL2) || SCR_EL3.FGTEn == '1')));
        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            exception.trappedsyscallinst = TRUE;
            preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

// AArch64.CheckForWFxTrap()
// ================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
    assert HaveEL(target_el);
    boolean is_wfe = wfxtype IN {WFxType_WFE, WFxType_WFET};
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WFxTrap(wfxtype, target_el);
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

// AArch64.CheckIllegalState()
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
if PSTATE.IL == '1' then
  route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_IllegalState);
  if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

// AArch64.MonitorModeTrap()
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;
exception = ExceptionSyndrome(Exception_Uncategorized);
if IsSecureEL2Enabled() then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

// AArch64.SystemAccessTrap()
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;
exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
// AArch64.SystemAccessTrapSyndrome()  
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)

ExceptionRecord exception;

bits(32) instr = instr_in;

case ec of

  when 0x0           // Trapped access due to unknown reason.
      exception = ExceptionSyndrome(Exception_Uncategorized);

  when 0x7           // Trapped access to SVE, Advance SIMD&FP system register.
      exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
      exception.syndrome<24:20> = ConditionSyndrome();

  when 0x18          // Trapped access to system register or system instruction.
      exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
      instr = ThisInstr();
      exception.syndrome<21:20> = instr<20:19>;    // Op0
      exception.syndrome<19:17> = instr<7:5>;      // Op2
      exception.syndrome<16:14> = instr<18:16>;    // Op1
      exception.syndrome<13:10> = instr<15:12>;    // CRn
      exception.syndrome<9:5>  = instr<4:0>;        // Rt
      exception.syndrome<4:1>  = instr<11:8>;       // CRm
      exception.syndrome<0>   = instr<21>;          // Direction

  when 0x19          // Trapped access to SVE System register
      exception = ExceptionSyndrome(Exception_SVEAccessTrap);

  otherwise
      Unreachable();

return exception;

// AArch64.UndefinedFault()  
// ========================

AArch64.UnderndefinedFault()

route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

bits(64) preferred_exception_return = ThisInstrAddr();

vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_Uncategorized);

if UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    case wfxtype of
        when WFxType_WFI
            exception.syndrome<1:0> = '00';
        when WFxType_WFE
            exception.syndrome<1:0> = '01';
        when WFxType_WFIT
            exception.syndrome<1:0> = '10';
            if HaveFeatWFxT2() then
                exception.syndrome<2> = '1';  // Register field is valid
                exception.syndrome<9:5> = ThisInstr()<4:0>;
            else
                exception.syndrome<2> = '0';  // Register field is invalid
        when WFxType_WFET
            exception.syndrome<1:0> = '11';
            if HaveFeatWFxT2() then
                exception.syndrome<2> = '1';  // Register field is valid
                exception.syndrome<9:5> = ThisInstr()<4:0>;
            else
                exception.syndrome<2> = '0';  // Register field is invalid
        if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

    // CheckFPAdvSIMDEnabled64()
    // ===========================
    // AArch64 instruction wrapper
    CheckFPAdvSIMDEnabled64()
        AArch64.CheckFPAdvSIMDEnabled();

Library pseudocode for aarch64/exceptions/traps/CheckFPEncabled64

    // CheckFPEncabled64()
    // =================
    // AArch64 instruction wrapper
    CheckFPEncabled64()
        AArch64.CheckFPEncabled();
Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

// CheckLDST64BEnabled()
// ===================================
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
booleans trap = FALSE;
ints(25) iss = ZeroExtend('10'); // 0x2
ints(2) target_el;

if PSTATE.EL == EL0 then
  if !IsInHost() then
    trap = SCTLR_EL1.EnALS == '0';
    target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
  else
    trap = SCTLR_EL2.EnALS == '0';
    target_el = EL2;
else
  target_el = EL1;

if (!trap && EL2Enabled() && HaveFeatHCX() &&
((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
  trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
  target_el = EL2;

if trap then LDST64BTrap(target_el, iss);

Library pseudocode for aarch64/exceptions/traps/CheckST64BV0enabled

// CheckST64BV0Enabled()
// =====================
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
booleans trap = FALSE;
ints(25) iss = ZeroExtend('1'); // 0x1
ints(2) target_el;

if PSTATE.EL == EL0 then
  if !IsInHost() then
    trap = SCTLR_EL1.EnAS0 == '0';
    target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
  else
    trap = SCTLR_EL2.EnAS0 == '0';
    target_el = EL2;
else
  target_el = EL1;

if (!trap && EL2Enabled() && HaveFeatHCX() &&
((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
  trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
  target_el = EL2;

if !trap && PSTATE.EL != EL3 then
  trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
  target_el = EL3;

if trap then LDST64BTrap(target_el, iss);
Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
  boolean trap = FALSE;
  bits(25) iss = Zeros();
  bits(2) target_el;

  if PSTATE.EL == EL0 then
    if !IsInHost() then
      trap = SCTLR_EL1.EnASR == '0';
      target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
    else
      trap = SCTLR_EL2.EnASR == '0';
      target_el = EL2;
  else
    trap = EL2Enabled() && HCR_EL2.EnASR == '0';
    target_el = EL2;

  if (!trap && EL2Enabled() && HaveFeatHCX() &&
    ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
    trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
    target_el = EL2;

  if trap then LDST64BTrap(target_el, iss);

Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

// LDST64BTrap()
// =============
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_LDST64BTrap);
  exception.syndrome = iss;
  AArch64_TakeException(target_el, exception, preferred_exception_return, vect_offset);

  return;
Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```c
// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)

boolean delay_enabled;
integer delay;
case target_el of
  when EL1
    if !IsInHost() then 
      delay_enabled = SCTLR_EL1.TWEDEn == '1';
      delay = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
    else 
      delay_enabled = SCTLR_EL2.TWEDEn == '1';
      delay = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
  when EL2
    assert EL2Enabled();
    delay_enabled = HCR_EL2.TWEDEn == '1';
    delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
  when EL3
    delay_enabled = SCR_EL3.TWEDEn == '1';
    delay = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

return (delay_enabled, delay);
```

Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```c
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```
Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```c
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.
<bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;
    bits(25) iss = Zeros();
    bits(5) iss2 = Zeros();
    if !HaveFeatLS64() && HaveRASExt() && IsAsyncAbort(fault) then
        iss<12:11> = fault.errortype; // SET
    if d_side then
        if HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 then
            if (fault.statuscode IN {Fault_AccessFlag,
                                      Fault_Translation, Fault_Permission}) then
                (iss2, iss<24:14>, iss<12:11>) = LS64InstructionSyndrome();
            else
                if (IsSecondStage(fault) && !fault.s2fs1walk &&
                    (!HaveRASExt() && fault.acctype == AccType_TTW &&
                     boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) then
                    is<24:14> = LSInstructionSyndrome();
            if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
                iss<13> = '1'; // Fault is generated by use of VNCR_EL2
            if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
                iss<8> = '1'; iss<6> = '1';
            else
                iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
        return (iss, iss2);
```

Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome

```c
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field, and LST.
<bits(5), bits(11), bits(2)) LS64InstructionSyndrome();
```
Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```c
// AArch64.DataMemZero()
// =====================
// Write Zero to data memory

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AddressDescriptor memaddrdesc_in, integer size)
  iswrite = TRUE;
  AddressDescriptor memaddrdesc = memaddrdesc_in;
  for i = 0 to size-1
    accdesc = CreateAccessDescriptor(AccType_DCZVA);
    if HaveMTEExt() then
      if AArch64.AccessIsTagChecked(vaddress, AccType_DCZVA) then
        bits(4) ptag = AArch64.PhysicalTag(vaddress);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
          if boolean IMPLEMENTATION_DEFINED "DC ZVA tag fault reported with lowest faulting address" then
            AArch64.TagCheckFault(vaddress, AccType_DCZVA, iswrite);
          else
            AArch64.TagCheckFault(regval, AccType_DCZVA, iswrite);
      else
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros());
      end
      if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
      end
      memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    end
  end
  return;
```

Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```c
// AArch64.TagMemZero()
// ====================
// Write Zero to tag memory

AArch64.TagMemZero(bits(64) vaddress_in, integer size)
  bits(64) vaddress = vaddress_in;
  integer count = size >> LOG2_TAG_GRANULE;
  bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
  for i = 0 to count-1
    AArch64.MemTag[vaddress, AccType_NORMAL] = tag;
    vaddress = vaddress + TAG_GRANULE;
  end
  return;
```
// AArch64.ExclusiveMonitorsPass()
// ===============================================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = TRUE;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());
    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    return passed;

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
// // It is permitted (but not required) for this function to return FALSE and
// // cause a store exclusive to fail if the virtual address region is not
// // totally included within the region recorded by MarkExclusiveVA().
// // It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```c
// AArch64.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(address, size)
    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    if IsFault(memaddrdesc) then
        return;
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
        AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```c
// FPRSqrtStepFused()
// =========

bits(N) FPRSqrtStepFused(op1_in, op2)
    assert N IN {16, 32, 64};
    result;
    op1 = op1_in;
    done;
    fpcr = FPCR[];
    op1 = FPNeg(op1);
    fpcr = !altfp && fpcr.AH == '1';
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode = '00';
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
    rounding = FPRoundingMode(fpcr);
    if !done then
        if (inf1 || zero2) || (zero1 || inf2) then
            result = FPOnePointFive('0');
        else
            result = FPOnePointFive('0');
        else
            result = FPInfinity(sign1 EOR sign2);
    else
        // Fully fused multiply-add and halve
        halve_result = (3.0 + (value1 * value2)) / 2.0;
        if halve_result == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign);
        else
            result = FPRound(halve_result, fpcr, rounding, fpexc);
    return result;
```
Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';             // Use RNE rounding mode
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);
            end
        end
    return result;
Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
if PSTATE.M<4> == '1' then return FALSE;
if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
  return FALSE;
if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
  return FALSE;
if !AArch64.AllocationTagAccessIsEnabled(acctype) then
  return FALSE;
if acctype IN {AccType_IFETCH, AccType_TTW, AccType_DC, AccType_IC} then
  return FALSE;
if acctype == AccType_NV2REGISTER then
  return FALSE;
if PSTATE.TCO=='1' then
  return FALSE;
if !IsTagCheckedInstruction() then
  return FALSE;
return TRUE;

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits(4) allocation_tag)
bits(64) result = address;
bits(4) tag;
if AArch64.AllocationTagAccessIsEnabled(acctype) then
  tag = allocation_tag;
else
  tag = '0000';
result<59:56> = tag;
return result;

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
return tagged_address<59:56>;
Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype, boolean iswrite)

aligned = (address == Align(address, alignment));
atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
vector = acctype == AccType_VEC;
boolean check;
if SCTLR[].A == '1' then check = TRUE;
elseif HaveLSE2Ext() then
    check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
else check = atomic || ordered;

if check && !aligned then
    secondstage = FALSE;
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
    return aligned;

Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

// AArch64.CheckTag()
// ==================

// Performs a Tag Check operation for a memory access and returns whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ptag, boolean write)

if memaddrdesc.memattrs.tagged then
    (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then  
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return ptag == readtag;
else
    return TRUE;
// AArch64.MemSingle[] - non-assignment (read) form
// ====================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] boolean ispair = FALSE;
    return AArch64.MemSingle[address, size, acctype, aligned, ispair];

// AArch64.MemSingle[] - non-assignment (read) form
// ====================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);
    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
        (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite, PhysMemRetStatus memstatus);
        if atomic then
            (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
        elsif splitpair then
            assert ispair;
            (memstatus, lowhalf, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
            (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
            value = highhalf:lowhalf;
        else
            for i = 0 to size-1
                (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
                if IsFault(memstatus) then
                    HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            return value;
    // AArch64.MemSingle[] - assignment (write) form
    // ==============================================================

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value;
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;
// AArch64.MemSingle[] - assignment (write) form
// =============================================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits(size*8) value

assert size IN {1, 2, 4, 8, 16};
constant halFSIZE = size DIV 2;
if HaveLSE2Ext() then
    assert CheckAllInAlignedQuantity(address, size, 16);
else
    assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

(atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite);
if atomic then
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        bits(halFSIZE*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;

        memstatus = PhysMemWrite(memaddrdesc, halFSIZE, accdesc, lowhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halFSIZE, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halFSIZE;
        memstatus = PhysMemWrite(memaddrdesc, halFSIZE, accdesc, highhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halFSIZE, accdesc);
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;
// AArch64.MemTag[] - non-assignment (read) form
// =============================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccType acctype]
AddressDescriptor memaddrdesc;
bits(4) value;

iswrite = FALSE;
aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, TAG_GRANULE);
accdesc = CreateAccessDescriptor(acctype);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

return tag;
else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// ==========================================
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
AddressDescriptor memaddrdesc;
iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, TAG_GRANULE);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType_Device then
    c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
    assert c IN {Constraint_NONE, Constraint_FAULT};
    if c == Constraint_FAULT then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
accdesc = CreateAccessDescriptor(acctype);

// Memory array access
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```c
// AArch64.PhysicalTag()
// =====================
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```c
// AArch64.TranslateAddressForAtomicAccess()
// =========================================
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;
    assert size IN {1, 2, 4, 8, 16};
    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite, aligned, size);

    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);

    return memaddrdesc;
```

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```c
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.

boolean AddressSupportsLS64(bits(64) address);
```

Library pseudocode for aarch64/functions/memory/CheckAllInAlignedQuantity

```c
// CheckAllInAlignedQuantity()
// ===========================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return Align(address+size-1, alignment) == Align(address, alignment);
```
Library pseudocode for aarch64/functions/memory/CheckSPAlignment

// CheckSPAlignment()
// ================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
// CheckSingleAccessAttributes()
// =============================
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED UNPREDICTABLE
// if they generate an alignment fault. If they do not generate an alignment fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
// The function returns (atomic, splitpair), where
// atomic indicates if the access is single copy atomic.
// splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
// when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
// as byte accesses.

(boolean, boolean) CheckSingleAccessAttributes(bits(64) address, MemoryAttributes memattrs, integer size,
AccType acctype, boolean iswrite, boolean aligned, boolean ispair)

isnormalwb = (memattrs.memtype     == MemType_Normal&&
memattrs.inner.attrs == MemAttr_WB&&
memattrs.outer.attrs == MemAttr_WB);

atomic    = TRUE;
splitpair = FALSE;
if isnormalwb then return (atomic, splitpair);

accatomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
AccType_ATOMICORDEREDATOMIC, AccType_ATOMICLS64, AccType_A32LSMD};

ordered    = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC,
AccType_ORDEREDATOMICRW, AccType_ATOMICORDEREDATOMIC, AccType_ATOMICLS64, AccType_A32LSMD};

if !aligned && (accatomic || ordered) then
                atomic = ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC);
                if atomic then
                    secondstage = FALSE;
                    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
                else
                    return (atomic, splitpair);
            
if ispair && aligned then
                // load / store pair requests that are aligned to each register access are split into 2 single copy atomic accesses
                atomic    = FALSE;
                splitpair = TRUE;
                return (atomic, splitpair);

if aligned then
                return (atomic, splitpair);

atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not Normal Inner Write-Back are Atomic";
return (atomic, splitpair);

Library pseudocode for aarch64/functions/memory/IsTagCheckedInstruction

// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.

boolean IsTagCheckedInstruction();
MEM{} - non-assignment (read) form
==================================
Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
Instruction fetches would call AArch64.MemSingle directly.

\[
\text{bits(size*8) Mem[bits(64) address, integer size, }\text{ AccType acctype]}
\]

\[
\text{boolean ispair = FALSE;}
\]

\[
\text{return Mem[address, size, acctype, ispair];}
\]

\[
\text{bits(size*8) Mem[bits(64) address, integer size, }\text{ AccType acctype, boolean ispair]}
\]

\[
\text{assert size IN \{1, 2, 4, 8, 16\};}
\]

\[
\text{constant halfsize = size DIV 2;}
\]

\[
\text{bits(size * 8) value;}
\]

\[
\text{bits(halfsize * 8) lowhalf, highhalf;}
\]

\[
\text{boolean iswrite = FALSE;}
\]

\[
\text{boolean aligned;}
\]

\[
\text{if ispair then}
\]

\[
\text{// check alignment on size of element accessed, not overall access size}
\]

\[
\text{aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);}
\]

\[
\text{else}
\]

\[
\text{aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);}
\]

\[
\text{boolean atomic;}
\]

\[
\text{if size != 16 || !(acctype IN \{AccType_VEC, AccType_VECSTREAM\}) then}
\]

\[
\text{if !HaveLSE2Ext() then}
\]

\[
\text{atomic = aligned;}
\]

\[
\text{else}
\]

\[
\text{atomic = CheckAllInAlignedQuantity(address, size, 16);}
\]

\[
\text{elsif acctype IN \{AccType_VEC, AccType_VECSTREAM\} then}
\]

\[
\text{// 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses}
\]

\[
\text{// 64-bit aligned.}
\]

\[
\text{atomic = address == Align(address, 8);}
\]

\[
\text{else}
\]

\[
\text{// 16-byte integer access}
\]

\[
\text{atomic = address == Align(address, 16);}
\]

\[
\text{if !atomic && ispair & address == Align(address, halfsize) then}
\]

\[
\text{single_is_pair = FALSE;}
\]

\[
\text{single_is_aligned = TRUE;}
\]

\[
\text{lowhalf = AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, single_is_pair];}
\]

\[
\text{highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, single_is_pair];}
\]

\[
\text{value = highhalf:lowhalf;}
\]

\[
\text{elsif atomic & ispair then}
\]

\[
\text{value = AArch64.MemSingle[address, size, acctype, aligned, ispair];}
\]

\[
\text{elsif !atomic then}
\]

\[
\text{assert size > 1;}
\]

\[
\text{value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];}
\]

\[
\text{// For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory}
\]

\[
\text{// access will generate an Alignment Fault, as to get this far means the first byte did}
\]

\[
\text{// not, so we must be changing to a new translation page.}
\]

\[
\text{if aligned then}
\]

\[
\text{c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);}
\]

\[
\text{assert c IN \{Constraint_FAULT, Constraint_NONE\};}
\]

\[
\text{if c == Constraint_NONE then aligned = TRUE;}
\]

\[
\text{for i = 1 to size-1}
\]

\[
\text{value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];}
\]

\[
\text{elsif size == 16 && acctype IN \{AccType_VEC, AccType_VECSTREAM\} then}
\]

\[
\text{lowhalf = AArch64.MemSingle[address, halfsize, acctype, aligned, ispair];}
\]

\[
\text{highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair];}
\]

\[
\text{value = highhalf:lowhalf;}
\]

\[
\text{else}
\]

\[
\text{value = AArch64.MemSingle[address, size, acctype, aligned, ispair];}
\]

\[
\text{if BigEndian(acctype) then}
\]

\[
\text{value = BigEndianReverse(value);}
\]

\[
\text{return value;}
\]
// Mem[] - assignment (write) form
// ---------------------------------
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value_in
boolean ispair = FALSE;
Mem[address, size, acctype, ispair] = value_in;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
boolean iswrite = TRUE;
constant halfsize = size DIV 2;
bits(size*8) value = value_in;
bits(halfsize*8) lowhalf, highhalf;
boolean atomic;
boolean aligned;
if BigEndian(acctype) then
    value = BigEndianReverse(value);
else
    atomic = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
        if !Atomic(acctype) then
            atomic = CheckAllInAlignedQuantity(address, size, 16);
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
        end
    elsif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
        end
    end
elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
    <highhalf, lowhalf> = value;
    AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
    AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
else
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
end
if !atomic && ispair && address == Align(address, 16);
if atomic && ispair && address == Align(address, 16);
if !atomic then
    assert size > 1;
    AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;
    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
    // access will generate an Alignment Fault, as to get this far means the first byte did
    // not, so we must be changing to a new translation page.
    for i = 1 to size-1
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    end
else
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
return;
Library pseudocode for aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype)
bits(size) newvalue;
memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
ldaccdesc = CreateAccessDescriptor(ldacctype);
staccdesc = CreateAccessDescriptor(stacctype);

// All observers in the shareability domain observe the
// following load and store atomically.
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
if BigEndian(ldacctype) then
    oldvalue = BigEndianReverse(oldvalue);

    case op of
        when MemAtomicOp_ADD newvalue = oldvalue + value;
        when MemAtomicOp_BIC newvalue = oldvalue AND NOT(value);
        when MemAtomicOp_EOR newvalue = oldvalue EOR value;
        when MemAtomicOp_ORR newvalue = oldvalue OR value;
        when MemAtomicOp_SMAX newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
        when MemAtomicOp_SMIN newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
        when MemAtomicOp_UMAX newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
        when MemAtomicOp_UMIN newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
        when MemAtomicOp_SWP newvalue = value;
    end;

    if BigEndian(stacctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);

    // Load operations return the old (pre-operation) value
    return oldvalue;
```

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/memory/MemAtomicCompareAndSwap

```plaintext
// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                  bits(size) newvalue_in, AccType ldacctype, AccType stacctype) {
  bits(size) newvalue = newvalue_in;
  memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
  ldaccdesc = CreateAccessDescriptor(ldacctype);
  staccdesc = CreateAccessDescriptor(stacctype);

  // All observers in the shareability domain observe the
  // following load and store atomically.
  (memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
  if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
  if BigEndian(ldacctype) then
    oldvalue = BigEndianReverse(oldvalue);
  if oldvalue == expectedvalue then
    if BigEndian(stacctype) then
      newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if IsFault(memstatus) then
      HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);
  return oldvalue;
}
```
Library pseudocode for aarch64/functions/memory/MemLoad64B

// MemLoad64B()
//============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
  bits(512) data;
  boolean iswrite = FALSE;
  constant integer size = 64;

  aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

  if ! AddressSupportsLs64(address) then
    c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
    assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

  if c == Constraint_FAULT then
    // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
    boolean secondstage = FALSE;
    boolean s2fs1walk = FALSE;
    fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fs1walk);
    AArch64.Abort(address, fault);
  else
    // Accesses are not single-copy atomic above the byte level
    for i = 0 to 63
      data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
  return data;

AddressDescriptor memaddrdesc;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
  ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTE2Ext() then
  if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if ! AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
      AArch64.TagCheckFault(address, acctype, iswrite);

PhysMemRetStatus memstatus;
(memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
  HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
return data;
Library pseudocode for aarch64/functions/memory/MemStore64B

```
// MemStore64B()
// ============
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    boolean iswrite = TRUE;
    constant integer size = 64;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fs1walk = FALSE;
            fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fs1walk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
        end if

    else
        MemStore64BWithRet(address, value, acctype); // Return status is ignored by ST64B
    end if
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```
// MemStore64BWithRet()
// ===============
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype)
    AddressDescriptor memaddrdesc;
    boolean iswrite = TRUE;
    constant integer size = 64;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1');
    end if

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NS then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    end if

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                AArch64.TagCheckFault(address, acctype, iswrite);
                return ZeroExtend('1');
            end if
        end if
    end if

    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    end if
    return memstatus.store64bstatus;
```
Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.
bits(64) MemStore64BWithRetStatus();

Library pseudocode for aarch64/functions/memory/NVMem

// NVMem[] - non-assignment form
// ================
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
//   supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// ================
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
//   supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;

Library pseudocode for aarch64/functions/memory/PhysMemTagRead

// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);

Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);
// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);

// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

// Returns the size of the block this performed for an iteration of the copy given the the
// parameters of the destination, source and size of the copy.
integer CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

// Check for EL0 and EL1 access to the CPY* and SET* instructions.
CheckMOPSEnabled()

MaxBlockSizeCopiedBytes()

enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };

integer MaxBlockSizeCopiedBytes()

return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";
// MemCpyAccessTypes()  
// ===================  
// Return the read and write access types for a CPY* instruction.

(AccType, AccType) MemCpyAccessTypes(bits(4) options)
    unpriv_at_el1 = PSTATE.EL == EL1 && !((EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';
    runpriv_at_el1 = options<1> == '1' && unpriv_at_el1;
    runpriv_at_el2 = options<1> == '1' && unpriv_at_el2;
    wunpriv_at_el1 = options<0> == '1' && unpriv_at_el1;
    wunpriv_at_el2 = options<0> == '1' && unpriv_at_el2;
    user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
    AccType racctype;
        if !user_access_override && (runpriv_at_el1 || runpriv_at_el2) then racctype = if options<3> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
        else racctype = if options<3> == '0' then AccType_NORMAL else AccType_STREAM;
    AccType wacctype;
        if !user_access_override && (wunpriv_at_el1 || wunpriv_at_el2) then wacctype = if options<2> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
        else wacctype = if options<2> == '0' then AccType_NORMAL else AccType_STREAM;
    return (racctype, wacctype);

Library pseudocode for aarch64/functions/mops/MemCpyDirectionChoice

// Returns true if in the non-overlapping case of a memcpy of size cpysize bytes
// from the source address fromaddress to destination address toaddress is done
// in the forward direction on this implementation.
boolean MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize);

Library pseudocode for aarch64/functions/mops/MemCpyOptionA

// MemCpyOptionA()  
// ===============  
// Returns TRUE if the implementation uses Option A for the
// CPY*/SET* instructions, and FALSE otherwise.

boolean MemCpyOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY*/SET* instructions use Option A";

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedE

// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedM

// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParameters IllformedM(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemCpyZeroSizeCheck();

Library pseudocode for aarch64/functions/mops/MemSetAccessType

// MemSetAccessType()
// ==================
// Return the access type for a SET* instruction.

AccType MemSetAccessType(bits(2) options)

unpriv_at_el1 = options<0> == '1' && PSTATE.EL == EL1 && !(EL2Enabled() &&
  HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = (options<0> == '1' && PSTATE.EL == EL2 &&
  HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');
user_access_override = HaveUAOExt() && PSTATE.UAO == '1';

AccType acctype;
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = if options<1> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
else
  acctype = if options<1> == '0' then AccType_NORMAL else AccType_STREAM;
return acctype;

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedE

// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.
boolean MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize,
  boolean IsSETGE);

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedM

// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.
boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize,
  boolean IsSETGM);

Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemSetZeroSizeCheck();
Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

// MismatchedCpySetTargetEL()
// ==========================
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;
    fi
    return target_el;

Library pseudocode for aarch64/functions/mops/MismatchedMemCpyException

// MismatchedMemCpyException()
// ===========================
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
    boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();
    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24> = '0';
    exception.syndrome<23> = '0';
    exception.syndrome<22:19> = options;
    exception.syndrome<18> = if from_epilogue then '1' else '0';
    exception.syndrome<17> = if wrong_option then '1' else '0';
    exception.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5> = srcreg<4:0>;
    exception.syndrome<4:0> = sizereg<4:0>;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/functions/mops/MismatchedMemSetException

// MismatchedMemSetException()
// ===========================
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer datareg, integer sizereg,
  boolean wrong_option, boolean from_epilogue, bits(2) options,
  boolean is_SETG)

  bits(64) preferred_exception_return = ThisInstrAddr();
  integer vect_offset = 0x0;
  bits(2) target_el = MismatchedCpySetTargetEL();

  ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
  exception.syndrome<24> = '1';
  exception.syndrome<23> = if is_SETG then '1' else '0';
  // exception.syndrome<22:21> is RES0
  exception.syndrome<20:19> = options;
  exception.syndrome<18> = if from_epilogue then '1' else '0';
  exception.syndrome<17> = if wrong_option then '1' else '0';
  exception.syndrome<16> = if option_a then '1' else '0';
  // exception.syndrome<15> is RES0
  exception.syndrome<14:10> = destreg<4:0>;
  exception.syndrome<9:5> = datareg<4:0>;
  exception.syndrome<4:0> = sizereg<4:0>;

  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/functions/mops/SETPostSizeChoice

// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGE);

Library pseudocode for aarch64/functions/mops/SETPreSizeChoice

// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);

Library pseudocode for aarch64/functions/mops/SETSizeChoice

// Returns the size of the block this performed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of AlignSize.
integer SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);
AddPAC()  
 ========  
 Calculates the pointer authentication code for a 64-bit quantity and then  
 inserts that into pointer authentication code field of that 64-bit quantity.

```
bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)  
bites(64) PAC;  
bites(64) result;  
bites(64) ext_ptr;  
bites(64) extfield;  
bit selbit;  
boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';  
integer top_bit = if tbi then 55 else 63;  

If tagged pointers are in use for a regime with two TTBRs, use bit<55> of  
the pointer to select between upper and lower ranges, and preserve this.  
This handles the awkward case where there is apparently no correct choice between  
the upper and lower address range - ie an addr of 1xxxxxxxx0... with TBI0=0 and TBI1=1  
and @xxxxxxxx1 with TBI1=0 and TBI0=1:
```

```
if PtrHasUpperAndLowerAddRanges() then  
  assert S1TranslationRegime() IN {EL1, EL2};  
  if S1TranslationRegime() == EL1 then  
    // EL1 translation regime registers  
    if data then  
      if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then  
        selbit = ptr<55>;  
      else  
        selbit = ptr<63>;  
    else  
      if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||  
       (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then  
        selbit = ptr<55>;  
      else  
        selbit = ptr<63>;  
    else  
      // EL2 translation regime registers  
      if data then  
        if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then  
          selbit = ptr<55>;  
        else  
          selbit = ptr<63>;  
      else  
        if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||  
         (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then  
          selbit = ptr<55>;  
        else  
          selbit = ptr<63>;  
      else selbit = if tbi then ptr<55> else ptr<63>;  
    if HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;  
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);  
  else  
    // The pointer authentication code field takes all the available bits in between  
    extfield = Replicate(selbit, 64);  

  // Compute the pointer authentication code for a ptr with good extension bits  
  if tbi then  
    ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;  
  else  
    ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;  

  PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);  

  // Check if the ptr has good extension bits and corrupt the pointer authentication code if not  
  if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then  
    if HaveEnhancedPAC() then  
      PAC = 0x0000000000000000<63:0>;  
    elseif !HaveEnhancedPAC2() then  
      PAC<top_bit-1> = NOT(PAC<top_bit-1>);  
```
// preserve the determination between upper and lower address at bit<55> and insert PAC
if !HaveEnhancedPAC2() then
  if tbi then
    result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
  else
    result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
else
  if tbi then
    result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
  else
    result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR
        PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
return result;

Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

// AddPACDA()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.
bits(64) AddPACDA(bits(64) X, bits(64) Y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APDAKey_EL1;

  APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = SITranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
      TrapEL2 = (EL2Enabled() & HCR_EL2.API == '0') &
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) & SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnDA;
      TrapEL2 = EL2Enabled() & HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) & SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnDA;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) & SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnDA;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;
  endcase
  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return AddPAC(X, Y, APDAKey_EL1, TRUE);
// AddPACDB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APDBKey_EL1;

  APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = S1TranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                  (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnDB;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnDB;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnDB;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;
  endcase
  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return AddPAC(X, Y, APDBKey_EL1, TRUE);
// AddPACGA()
// =========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(128) APGAKey_EL1;

APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' &&
             (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

if TrapEL2 then TrapPACUse(EL2);
elsif TrapEL3 then TrapPACUse(EL3);
else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:Zeros(32);
Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

// AddPACIA()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIAKey_EL1;

APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
              (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIA;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
if Enable == '0' then return X;
elseif TrapEL2 then TrapPACUse(EL2);
elseif TrapEL3 then TrapPACUse(EL3);
else return AddPAC(X, Y, APIAKey_EL1, FALSE);
Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIBKey_EL1;

APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
               (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3-APIBKey_EL1;
  when EL1
    Enable = SCTLR_EL1.EnIB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3-APIBKey_EL1;
  when EL2
    Enable = SCTLR_EL2.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3-APIBKey_EL1;
  when EL3
    Enable = SCTLR_EL3.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
if Enable == '0' then return X;
elsif TrapEL2 then TrapPACUse(EL2);
elsif TrapEL3 then TrapPACUse(EL3);
else return AddPAC(X, Y, APIBKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// ========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_PACFail);
exception syndrome<1:0> = syndrome;
exception syndrome<24:2> = Zeros(); // RES0
if UInt(PSTATE.EL) > UInt(EL0) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```
Library pseudocode for aarch64/functions/pac/auth/Auth

// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
    boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);
    if tbi then
        original_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
            else
                if !HaveEnhancedPAC2() then
                    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                        result = original_ptr;
                    else
                        error_code = key_number:NOT(key_number);
                        result = original_ptr<63>:error_code:original_ptr<60:0>;
                else
                    result = ptr;
                    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
                    result<63:56> = result<63:56> EOR PAC<63:56>;
                    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                        if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                            error_code = (if data then '1' else '0'):key_number;
                            AArch64.PACFailException(error_code);
        else
            result = result;
            return result;
Library pseudocode for aarch64/functions/pac/authda/AuthDA

// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APDAKey_EL1;

  APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = SITranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                     (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnDA;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnDA;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnDA;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);
Library pseudocode for `aarch64/functions/pac/authdb/AuthDB`

```
// AuthDB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
    end case
    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);
```

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/pac/authia/AuthIA

```c
bits(64) AuthIA(bits(64) X, bits(64) Y, boolean is_combined)

boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIAKey_EL1;

APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;

if Enable == '0' then return X;
elsif TrapEL2 then TrapPACUse(EL2);
elsif TrapEL3 then TrapPACUse(EL3);
else return Auth(X, Y, APIAKey_EL1, FALSE, '0', is_combined);
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

// AuthIB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y, boolean is_combined)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APIBKey_EL1;

  APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
  case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
              (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
  if Enable == '0' then return X;
  elsifTrapEL2 then TrapPACUse(EL2);
  elsifTrapEL3 then TrapPACUse(EL3);
  else return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
integer tsz_field;
boolean using64k;
if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
        // EL1 translation regime registers
        tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
        using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
    else
        // EL2 translation regime registers
        assert HaveEL(EL2);
        tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
        using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    end;
else
    tksz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
    using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';
end;

max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
if tsz_field > max_limit_tsz_field then
    // TCR_ELx.TySZ is out of range
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_NONE};
    tszmin = if using64k && AArch64.VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    end;
return (64-tsz_field);
// ComputePAC()
// ============
bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)

bits(64) workingval;
bits(64) runningmod;
bits(64) roundkey;
bits(64) modk0;
constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

integer iterations;
if HavePACQARMA3() then
  iterations = 2;
  RC[0] = 0x0000000000000000<63:0>;
  RC[1] = 0x131982E0D737344<63:0>;
  RC[2] = 0xA4093822299F31D0<63:0>;
else
  iterations = 4;
  RC[0] = 0x0000000000000000<63:0>;
  RC[1] = 0x131982E0D737344<63:0>;
  RC[2] = 0xA4093822299F31D0<63:0>;
  RC[3] = 0x082EFA08EC4E6C89<63:0>;
  RC[4] = 0x452821E638DD01377<63:0>;
end if;

modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
runningmod = modifier;
workingval = data EOR key0;
for i = 0 to iterations
  roundkey = key1 EOR runningmod;
  workingval = workingval EOR roundkey;
  workingval = workingval EOR RC[i];
  if i > 0 then
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if HavePACQARMA3() then
      workingval = PACSub1(workingval);
    else
      workingval = PACSub(workingval);
    end if;
    runningmod = TweakShuffle(runningmod<63:0>);
  end if;
end for;
roundkey = modk0 EOR runningmod;
workingval = workingval EOR roundkey;
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
if HavePACQARMA3() then
  workingval = PACSub1(workingval);
else
  workingval = PACSub(workingval);
end if;
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
workingval = key1 EOR workingval;
workingval = PACCellInvShuffle(workingval);
if HavePACQARMA3() then
  workingval = PACSub1(workingval);
else
  workingval = PACSub1(workingval);
end if;
workingval = PACMult(workingval);
workingval = PACCellInvShuffle(workingval);
for i = 0 to iterations
  if HavePACQARMA3() then
    workingval = PACSub1(workingval);
  else
    workingval = PACInvSub(workingval);
  end if;
  if i < iterations then
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    runningmod = TweakInvShuffle(runningmod<63:0>);
    roundkey = key1 EOR runningmod;
  end if;
end for;
workingval = workingval EOR RC[iterations-i];
workingval = workingval EOR roundkey;
workingval = workingval EOR Alpha;
workingval = workingval EOR modk0;
return workingval;

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

// PACCellInvShuffle()
// =============

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

// PACCellShuffle()
// =============

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```c
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000'  Toutput<4*i+3:4*i> = '0101';
        when '0001'  Toutput<4*i+3:4*i> = '1110';
        when '0010'  Toutput<4*i+3:4*i> = '1101';
        when '0011'  Toutput<4*i+3:4*i> = '1000';
        when '0100'  Toutput<4*i+3:4*i> = '1010';
        when '0101'  Toutput<4*i+3:4*i> = '1011';
        when '0110'  Toutput<4*i+3:4*i> = '0001';
        when '0111'  Toutput<4*i+3:4*i> = '1001';
        when '1000'  Toutput<4*i+3:4*i> = '0010';
        when '1001'  Toutput<4*i+3:4*i> = '0110';
        when '1010'  Toutput<4*i+3:4*i> = '1111';
        when '1011'  Toutput<4*i+3:4*i> = '0000';
        when '1100'  Toutput<4*i+3:4*i> = '0100';
        when '1101'  Toutput<4*i+3:4*i> = '1100';
        when '1110'  Toutput<4*i+3:4*i> = '0111';
        when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```c
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
bits(4)  t0;
bits(4)  t1;
bits(4)  t2;
bits(4)  t3;
bits(64) Soutput;
for i = 0 to 3
    t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
    t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
    t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
    t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
    t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
    t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    Soutput<4*i+3:4*i> = t3<3:0>;
    Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
    Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
    Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
return Soutput;
```
Library pseudocode for aarch64/functions/pac/computepac/PACSub

// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
   // This is a 4-bit substitution from the PRINCE-family cipher
   bits(64) Toutput;
   for i = 0 to 15
      case Tinput<4*i+3:4*i> of
         when '0000'  Toutput<4*i+3:4*i> = '1011';
         when '0001'  Toutput<4*i+3:4*i> = '0110';
         when '0010'  Toutput<4*i+3:4*i> = '1000';
         when '0011'  Toutput<4*i+3:4*i> = '1111';
         when '0100'  Toutput<4*i+3:4*i> = '1100';
         when '0101'  Toutput<4*i+3:4*i> = '0000';
         when '0110'  Toutput<4*i+3:4*i> = '1001';
         when '0111'  Toutput<4*i+3:4*i> = '1110';
         when '1000'  Toutput<4*i+3:4*i> = '0011';
         when '1001'  Toutput<4*i+3:4*i> = '0111';
         when '1010'  Toutput<4*i+3:4*i> = '0100';
         when '1011'  Toutput<4*i+3:4*i> = '1101';
         when '1100'  Toutput<4*i+3:4*i> = '0001';
         when '1101'  Toutput<4*i+3:4*i> = '1010';
      return Toutput;

Library pseudocode for aarch64/functions/pac/computepac/PacSub1

// PacSub1()
// ========

bits(64) PACSub1(bits(64) Tinput)
   // This is a 4-bit substitution from Qarma sigma1
   bits(64) Toutput;
   for i = 0 to 15
      case Tinput<4*i+3:4*i> of
         when '0000' Toutput<4*i+3:4*i> = '1010';
         when '0001' Toutput<4*i+3:4*i> = '1101';
         when '0010' Toutput<4*i+3:4*i> = '1110';
         when '0011' Toutput<4*i+3:4*i> = '0110';
         when '0100' Toutput<4*i+3:4*i> = '1111';
         when '0101' Toutput<4*i+3:4*i> = '0111';
         when '0110' Toutput<4*i+3:4*i> = '0011';
         when '0111' Toutput<4*i+3:4*i> = '0101';
         when '1000' Toutput<4*i+3:4*i> = '1001';
         when '1001' Toutput<4*i+3:4*i> = '1000';
         when '1010' Toutput<4*i+3:4*i> = '0000';
         when '1011' Toutput<4*i+3:4*i> = '1100';
         when '1100' Toutput<4*i+3:4*i> = '1011';
         when '1101' Toutput<4*i+3:4*i> = '0001';
         when '1110' Toutput<4*i+3:4*i> = '0010';
         when '1111' Toutput<4*i+3:4*i> = '0100';
      return Toutput;

Library pseudocode for aarch64/functions/pac/computepac/RC

array bits(64) RC[0..4];
Library pseudocode for aarch64/functions/pac/computepac/RotCell

```c
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
  bits(8) tmp;
  bits(4) outcell;

  // assert amount>3 || amount<1;
  tmp<7:0> = incell<3:0>:incell<3:0>;
  outcell = tmp<7-amount:4-amount>;
  return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```c
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4) incell)
  bits(4) outcell;
  outcell<3> = incell<2>;
  outcell<2> = incell<1>;
  outcell<1> = incell<0>;
  outcell<0> = incell<0> EOR incell<3>;
  return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```c
// TweakCellRot()  
// =============

bits(4) TweakCellRot(bits(4) incell)
  bits(4) outcell;
  outcell<3> = incell<0> EOR incell<1>;
  outcell<2> = incell<3>;
  outcell<1> = incell<2>;
  outcell<0> = incell<1>;
  return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```c
// TweakInvShuffle()
// ===============

bits(64) TweakInvShuffle(bits(64) indata)
  bits(64) outdata;
  outdata<3:0> = TweakCellInvRot(indata<51:48>);
  outdata<7:4> = indata<55:52>;
  outdata<11:8> = indata<23:20>;
  outdata<15:12> = indata<27:24>;
  outdata<19:16> = indata<3:0>;
  outdata<23:20> = indata<7:4>;
  outdata<27:24> = TweakCellInvRot(indata<11:8>);
  outdata<31:28> = indata<15:12>;
  outdata<35:32> = TweakCellInvRot(indata<31:28>);
  outdata<39:36> = TweakCellInvRot(indata<63:60>);
  outdata<43:40> = TweakCellInvRot(indata<59:56>);
  outdata<47:44> = TweakCellInvRot(indata<19:16>);
  outdata<51:48> = indata<35:32>;
  outdata<55:52> = indata<39:36>;
  outdata<59:56> = indata<43:40>;
  outdata<63:60> = TweakCellInvRot(indata<47:44>);
  return outdata;
```
Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```c
// TweakShuffle()
// ==============
bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
return outdata;
```

Library pseudocode for aarch64/functions/pac/pac/ConstPACField

```c
// ConstPACField()
// ===============
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.
boolean ConstPACField() {
    return boolean IMPLEMENTATION_DEFINED "Bit 55 determines the size of the PAC field";
}
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```c
// HaveEnhancedPAC()
// ===============
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.
boolean HaveEnhancedPAC() {
    return ( HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
}
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

```c
// HaveEnhancedPAC2()
// ===============
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.
boolean HaveEnhancedPAC2() {
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC 2 functionality" );
}
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```c
// HaveFPAC()
// ============
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.
boolean HaveFPAC() {
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
}
```
// HaveFPACCombined()
// ===============
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";

// HavePACExt()
// ============
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);

// HavePACIMP()
// ============
// Returns TRUE if support for PAC IMP is implemented, FALSE otherwise.

boolean HavePACIMP()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC IMP functionality";

// HavePACQARMA3()
// ===============
// Returns TRUE if support for PAC QARMA3 is implemented, FALSE otherwise.

boolean HavePACQARMA3()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 functionality";

// HavePACQARMA5()
// ===============
// Returns TRUE if support for PAC QARMA5 is implemented, FALSE otherwise.

boolean HavePACQARMA5()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 functionality";

// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL, AccType_NORMAL);
    return HasUnprivileged(regime);
// Strip()
// ========
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
  bits(64) original_ptr;
  bits(64) extfield;
  boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
  integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
  extfield = Replicate(A<55>, 64);
  if tbi then
    original_ptr = A<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;
  else
    original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;
  return original_ptr;

// TrapPACUse()
// ===========
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
  assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
  bits(64) preferred_exception_return = ThisInstrAddr();
  ExceptionRecord exception;
  vect_offset = 0;
  exception = Exception Syndrome( Exception_PACTrap);
  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    boolean mask_active;
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;
    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
        PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError();  // Set ISR_EL1.A to 0
        return;

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);

Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

// AArch64.ReportDeferredSError()
// ==============================
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1';  // A
    target<24> = syndrome<24>;  // IDS
    target<23:0> = syndrome<23:0>;  // ISS
    return target;
Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```plaintext
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
  assert PSTATE.EL IN {EL0, EL1} & EL2Enabled();

  // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
  // SError interrupt might be pending
  vSEI_enabled = HCR_EL2.TGE == '0' & HCR_EL2.AMO == '1';
  vSEI_pending = vSEI_enabled & HCR_EL2.VSE == '1';
  vintdis = Halted() || ExternalDebugInterrupsDisabled(EL1);
  vmasked = vintdis || PSTATE.A == '1';

  // Check for a masked virtual SError pending
  if vSEI_pending & vmasked then
    // This function might be called for the interworking case, and INTdis is masking
    // the virtual SError interrupt.
    if ELUsingAArch32(EL1) then
      VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
    else
      VDISR_EL2 = AArch64.ReportDeferredSError(VSES_EL2<24:0>);
    HCR_EL2.VSE = '0';
  return;
```

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```plaintext
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
  assert UsingAArch32(); // Always called from AArch32 state before entering AArch64 state

  integer first; integer last; boolean include_R15;
  if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
    first = 0; last = 14; include_R15 = FALSE;
  elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() & !ELUsingAArch32(EL2) then
    first = 0; last = 30; include_R15 = FALSE;
  else
    first = 0; last = 30; include_R15 = TRUE;
  for n = first to last
    if (n != 15 || include_R15) & ConstrainUnpredictableBool(Unpredictable ZEROUPPER) then
      _R[n]<63:32> = Zeros();
  return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```plaintext
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()
  for i = 0 to 30
    X[i] = bits(64) UNKNOWN;
  return;
```
// AArch64.ResetSIMDFPRegisters()
// ---------------------------------
AArch64.ResetSIMDFPRegisters()
  for i = 0 to 31
    V[i] = bits(128) UNKNOWN;
  return;

// AArch64.ResetSpecialRegisters()
// --------------------------------
AArch64.ResetSpecialRegisters()
  // AArch64 special registers
  SP_EL0 = bits(64) UNKNOWN;
  SP_EL1 = bits(64) UNKNOWN;
  SPSR_EL1 = bits(64) UNKNOWN;
  ELR_EL1 = bits(64) UNKNOWN;
  if HaveEL(EL2) then
    SP_EL2 = bits(64) UNKNOWN;
    SPSR_EL2 = bits(64) UNKNOWN;
    ELR_EL2 = bits(64) UNKNOWN;
  if HaveEL(EL3) then
    SP_EL3 = bits(64) UNKNOWN;
    SPSR_EL3 = bits(64) UNKNOWN;
    ELR_EL3 = bits(64) UNKNOWN;
  // AArch32 special registers that are not architecturally mapped to AArch64 registers
  if HaveAArch32EL(EL1) then
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
  // External debug special registers
  DLR_EL0 = bits(64) UNKNOWN;
  DSPSR_EL0 = bits(64) UNKNOWN;
  return;

AArch64.ResetSystemRegisters(boolean cold_reset);

// PC - non-assignment form
// ------------------------
// Read program counter.
bits(64) PC[]
  return _PC;
Library pseudocode for aarch64/functions/registers/SP

// SP[] - assignment form
// ===============
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
assert width IN {32,64};
if PSTATE.SP == '0' then
    SP_EL0 = ZeroExtend(value);
else
    case PSTATE.EL of
        when EL0 SP_EL0 = ZeroExtend(value);
        when EL1 SP_EL1 = ZeroExtend(value);
        when EL2 SP_EL2 = ZeroExtend(value);
        when EL3 SP_EL3 = ZeroExtend(value);
    return;

// SP[] - non-assignment form
// =========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
assert width IN {8,16,32,64};
if PSTATE.SP == '0' then
    return SP_EL0<width-1:0>;
else
    case PSTATE.EL of
        when EL0 return SP_EL0<width-1:0>;
        when EL1 return SP_EL1<width-1:0>;
        when EL2 return SP_EL2<width-1:0>;
        when EL3 return SP_EL3<width-1:0>;

Library pseudocode for aarch64/functions/registers/V

// V[] - assignment form
// ================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
assert n >= 0 && n < 32;
assert width IN {8,16,32,64,128};
integer vlen = if IsSVEEnabled(PSTATE.EL) then VL else 128;
if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
    _Z[n] = ZeroExtend(value);
else
    _Z[n]<vlen-1:0> = ZeroExtend(value);

// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
assert n >= 0 && n < 32;
assert width IN {8,16,32,64,128};
return _Z[n]<width-1:0>;
Library pseudocode for aarch64/functions/registers/Vpart

// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width) Vpart[integer n, integer part]
assert n >= 0 && n <= 31;
assert part IN {0, 1};
if part == 0 then
  assert width < 128;
  return V[n];
else
  assert width IN {32, 64};
  bits(128) vreg = V[n];
  return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part] = bits(width) value
assert n >= 0 && n <= 31;
assert part IN {0, 1};
if part == 0 then
  assert width < 128;
  V[n] = value;
else
  assert width == 64;
  bits(64) vreg = V[n];
  V[n] = value<63:0> : vreg;
// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);

// X[] - non-assignment form
// =========================
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
// AArch32.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAAArch32(EL1) then
        return AArch64.IsFPEnabled(el);
    if HaveEL(EL3) && ELUsingAAArch32(EL3) && 'IsSecure() then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;
    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        boolean disabled;
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
            when '11' disabled = FALSE;
        if disabled then return FALSE;
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAAArch32(EL2) then
            return AArch64.IsFPEnabled(EL2);
        if HCPTR.TCP10 == '1' then return FALSE;
    if HaveEL(EL3) && !ELUsingAAArch32(EL3) then
        // Check if access disabled in CPTR EL3
        if CPTR_EL3.TFP == '1' then return FALSE;
    return TRUE;
Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```java
// AArch64.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check FP&SIMD at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;
    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;
        // Check if access disabled in CPTR_EL3
        if HaveEL(EL3) then
            if CPTR_EL3.TFP == '1' then return FALSE;
        return TRUE;
```

Library pseudocode for aarch64/functions/sve/AnyActiveElement

```java
// AnyActiveElement()
// ==================
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

Library pseudocode for aarch64/functions/sve/CeilPow2

```java
// CeilPow2()
// =========
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```
// CheckSVEEnabled()
// CheckSVEEnabled()
// Check if access disabled in CPACR_EL1
// Check if access disabled in CPACR_EL1
// Check if access disabled in CPACR_EL1
bool disabled;
if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
    // Check if access disabled in CPACR_EL1
    // Check if access disabled in CPACR_EL1
    // Check if access disabled in CPACR_EL1
    case CPACR_EL1.ZEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    if disabled then SVEAccessTrap(EL1);
else
    if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
    if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
    // Check if access disabled in CPTR_EL3
    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then SVEAccessTrap(EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
Library pseudocode for aarch64/functions/sve/DecodePredCount

```c
// DecodePredCount()
// ----------------

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = VL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

Library pseudocode for aarch64/functions/sve/ElemFFR

```c
// ElemFFR[] - non-assignment form
// --------------------------------

bit ElemFFR[integer e, integer esize]
    return ElemP_FFR, e, esize];

// ElemFFR[] - assignment form
// ----------------------------

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= PL;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

Library pseudocode for aarch64/functions/sve/ElemP

```c
// ElemP[] - non-assignment form
// -----------------------------

bit ElemP[bits(N) pred, integer e, integer esize]
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

// ElemP[] - assignment form
// --------------------------

ElemP[bits(N) &pred, integer e, integer esize] = bit value
    assert esize IN {8, 16, 32, 64, 128};
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```
Library pseudocode for aarch64/functions/sve/FFR

// FFR[] - non-assignment form
// ================

bits(width) FFR[]
    assert width == PL;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// ================

FFR[] = bits(width) value
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(value);
    else
        _FFR<width-1:0> = value;

Library pseudocode for aarch64/functions/sve/FPCompareNE

// FPCompareNE()  
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    if op1_nan || op2_nan then
        result = TRUE;
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;

Library pseudocode for aarch64/functions/sve/FPCompareUN

// FPCompareUN()  
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;
Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// =============
bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, FPRounding rounding)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding);

// FPConvertSVE()
// =============
```

Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// =======
bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;
    return result;
```
bits(N) FPExpCoefficient(integer index)

assert N IN {16,32,64};
integer result;

if N == 16 then
    case index of
    when 0 result = 0x0000;
    when 1 result = 0x0016;
    when 2 result = 0x002d;
    when 3 result = 0x0045;
    when 4 result = 0x005d;
    when 5 result = 0x0075;
    when 6 result = 0x008e;
    when 7 result = 0x00a8;
    when 8 result = 0x00c2;
    when 9 result = 0x00dc;
    when 10 result = 0x00f8;
    when 11 result = 0x0114;
    when 12 result = 0x0130;
    when 13 result = 0x014d;
    when 14 result = 0x016b;
    when 15 result = 0x0189;
    when 16 result = 0x01a8;
    when 17 result = 0x01c8;
    when 18 result = 0x01e8;
    when 19 result = 0x0209;
    when 20 result = 0x022b;
    when 21 result = 0x024e;
    when 22 result = 0x0271;
    when 23 result = 0x0295;
    when 24 result = 0x02ba;
    when 25 result = 0x02e0;
    when 26 result = 0x0306;
    when 27 result = 0x032e;
    when 28 result = 0x0356;
    when 29 result = 0x037f;
    when 30 result = 0x03a9;
    when 31 result = 0x03d4;
elseif N == 32 then
    case index of
    when 0 result = 0x000000;
    when 1 result = 0x0164d2;
    when 2 result = 0x02cd87;
    when 3 result = 0x043a29;
    when 4 result = 0x05aac3;
    when 5 result = 0x071f62;
    when 6 result = 0x08980f;
    when 7 result = 0x0a14d5;
    when 8 result = 0x0b95c2;
    when 9 result = 0x0d1adf;
    when 10 result = 0x0ea43a;
    when 11 result = 0x1031dc;
    when 12 result = 0x11c3d3;
    when 13 result = 0x135a2b;
    when 14 result = 0x14f4f0;
    when 15 result = 0x16942d;
    when 16 result = 0x1837f0;
    when 17 result = 0x19e046;
    when 18 result = 0x1b8d3a;
    when 19 result = 0x1d3eda;
    when 20 result = 0x1ef532;
    when 21 result = 0x20b051;
    when 22 result = 0x227043;
    when 23 result = 0x243516;
    when 24 result = 0x25fed7;
    when 25 result = 0x27cd94;
when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3123f6;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fba9;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;
else // N == 64
  case index of
    when  0 result = 0x0000000000000000;
    when  1 result = 0x02c9a3e778061;
    when  2 result = 0x059b0d3158574;
    when  3 result = 0x0874518759bc8;
    when  4 result = 0x0b5586cf9890f;
    when  5 result = 0x0e3ec32d3d1a2;
    when  6 result = 0x11301d0125b51;
    when  7 result = 0x1429aaea92de0;
    when  8 result = 0x172b83c7d517b;
    when  9 result = 0x1a35b86fcb75;
    when 10 result = 0x1d4873168b9aa;
    when 11 result = 0x2063b8628cd6;
    when 12 result = 0x2387a6e756238;
    when 13 result = 0x26b4565e27cd0;
    when 14 result = 0x29e9df5f1f0ee1;
    when 15 result = 0x2d285a6e4030b;
    when 16 result = 0x306fe0a31b715;
    when 17 result = 0x33c0bb26416ff;
    when 18 result = 0x371a7373aa9cb;
    when 19 result = 0x3a7d0b34e59ff7;
    when 20 result = 0x3dea41c123422;
    when 21 result = 0x4160a21f72e2a;
    when 22 result = 0x44e0b6061892d;
    when 23 result = 0x48a2b5c13cd0;
    when 24 result = 0x4bfad5362a27;
    when 25 result = 0x4f9b769d2ca7;
    when 26 result = 0x5342b569d04f82;
    when 27 result = 0x56f4736b5270a;
    when 28 result = 0x5ab07dd485429;
When 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FE564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422AA0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B06DC5E5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA5503B23E255D;
when 47 result = 0xA9E6B5579F0BF;
when 48 result = 0xAE89F95AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F6F2FB5E47;
when 51 result = 0xBC1E994BC1D2;
when 52 result = 0xC199BD85529C;
when 53 result = 0xC67F12E57D148;
when 54 result = 0xCB720DCF9069;
when 55 result = 0xD07204A07897C;
when 56 result = 0xD58180CFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE7B3FF6;
when 60 result = 0xEA4FA2A4900A;
when 61 result = 0xEFA1BEE615A27;
when 62 result = 0xF50765B6E4548;
when 63 result = 0xFA7C1819E90D8;

return result<N-1:0>;

Library pseudocode for aarch64/functions/sve/FPMinNormal

// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Zeros(E-1):'1';
frac = Zeros(F);
return sign : exp : frac;

Library pseudocode for aarch64/functions/sve/FPOne

// FPOne()
// ========

bits(N) FPOne(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-1);
frac = Zeros(F);
return sign : exp : frac;
**Library pseudocode for aarch64/functions/sve/FPPointFive**

```plaintext
// FPPointFive()
// =============

bits(N) FPPointFive(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-2):'0';
frac = Zeros(F);
return sign : exp : frac;
```

**Library pseudocode for aarch64/functions/sve/FPProcess**

```plaintext
// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
bits(N) result;
assert N IN {16,32,64};
FPCRType fpcr = FPCR[];
(fptype,sign,value) = FPUnpack(input, fpcr);
if fptype == FPType_SNaN || fptype == FPType_QNaN then
result = FPProcessNaN(fptype, input, fpcr);
elsif fptype == FPType_Infinity then
result = FPInfinity(sign);
elsif fptype == FPType_Zero then
result = FPZero(sign);
else
result = FPRound(value, fpcr);
FPProcessDenorm(fptype, N, fpcr);
return result;
```

**Library pseudocode for aarch64/functions/sve/FPScale**

```plaintext
// FPScale()
// =========

bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
assert N IN {16,32,64};
bits(N) result;
(fptype,sign,value) = FPUnpack(op, fpcr);
if fptype == FPType_SNaN || fptype == FPType_QNaN then
result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPType_Zero then
result = FPZero(sign);
elsif fptype == FPType_Infinity then
result = FPInfinity(sign);
else
result = FPRound(value * (2.0^scale), fpcr);
FPProcessDenorm(fptype, N, fpcr);
return result;
```
Library pseudocode for aarch64/functions/sve/FPTrigMAdd

// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;
    if op2<\N-1> == '1' then
        x = x + 8;
    coeff = FPTrigMAddCoefficient[x];
    op2 = FPAbs(op2);
    result = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index]
assert N IN {16,32,64};
integer result;
if N == 16 then
    case index of
    when 0 result = 0x3c00;
    when 1 result = 0xb155;
    when 2 result = 0x2030;
    when 3 result = 0x0000;
    when 4 result = 0x0000;
    when 5 result = 0x0000;
    when 6 result = 0x0000;
    when 7 result = 0x0000;
    when 8 result = 0x3c00;
    when 9 result = 0xb800;
    when 10 result = 0x293a;
    when 11 result = 0x0000;
    when 12 result = 0x0000;
    when 13 result = 0x0000;
    when 14 result = 0x0000;
    when 15 result = 0x0000;
elsif N == 32 then
    case index of
    when 0 result = 0x3f800000;
    when 1 result = 0xbe2aaaab;
    when 2 result = 0x3c088886;
    when 3 result = 0xb95008b9;
    when 4 result = 0x36369d6d;
    when 5 result = 0x00000000;
    when 6 result = 0x00000000;
    when 7 result = 0x00000000;
    when 8 result = 0x3f800000;
    when 9 result = 0xbf000000;
    when 10 result = 0x3d2aaaa6;
    when 11 result = 0xbab60705;
    when 12 result = 0x37cd37cc;
    when 13 result = 0x00000000;
    when 14 result = 0x00000000;
    when 15 result = 0x00000000;
else // N == 64
    case index of
    when 0 result = 0x3ff0000000000000;
    when 1 result = 0xbfc555555555543;
    when 2 result = 0x3f8111111110f30c;
    when 3 result = 0xbf2a01a019b92fc6;
    when 4 result = 0x3ec71de351f3d22b;
    when 5 result = 0xbe5ae5e2b6f7b91;
    when 6 result = 0x3de58408868552f;
    when 7 result = 0x0000000000000000;
    when 8 result = 0x3ff0000000000000;
    when 9 result = 0xbfe0000000000000;
    when 10 result = 0x3fa5555555555536;
    when 11 result = 0xbf56c16c16c13a0b;
    when 12 result = 0x3efa01a019b1e8d8;
    when 13 result = 0xbe927e4f7282f468;
    when 14 result = 0xe21ee96d2641b13;
    when 15 result = 0xbda8f76380fb401;
    return result<N-1:0>;
Library pseudocode for aarch64/functions/sve/FPTrigSMul

// FPTrigSMul()
// ============

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
result = FPMul(op1, op1, fpcr);
fpexc = FALSE;
(type, sign, value) = FPUnpack(result, fpcr, fpexc);
if !(type IN {FPType_QNaN, FPType_SNaN}) then
    result<N-1> = op2<0>;
return result;

Library pseudocode for aarch64/functions/sve/FPTrigSSel

// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
assert N IN {16,32,64};
bits(N) result;
if op2<0> == '1' then
    result = FPOne(op2<1>);
elsif op2<1> == '1' then
    result = FPNeg(op1);
else
    result = op1;
return result;

Library pseudocode for aarch64/functions/sve/FirstActive

// FirstActive()
// =============

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';

Library pseudocode for aarch64/functions/sve/FloorPow2

// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);

Library pseudocode for aarch64/functions/sve/HaveSVE

// HaveSVE()
// =========

boolean HaveSVE()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

// HaveSVEFP32MatMulExt()
// ======================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";

Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

// HaveSVEFP64MatMulExt()
// ======================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits_in)
integer nbits = Min(nbits_in, MaxImplementedVL());
assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;
while nbits > 128 do
    if IsPow2(nbits) || SupportedNonPowerTwoVL(nbits) then return nbits;
    nbits = nbits - 128;
return nbits;

Library pseudocode for aarch64/functions/sve/IsEven

// IsEven()
// ========

boolean IsEven(integer val)
return val MOD 2 == 0;

Library pseudocode for aarch64/functions/sve/IsFPEnabled

// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point registers are enabled at the target exception level in the current execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
if ELUsingAArch32(el) then
    return AArch32.IsFPEnabled(el);
else
    return AArch64.IsFPEnabled(el);

Library pseudocode for aarch64/functions/sve/IsPow2

// IsPow2()
// =======
// Return TRUE if positive integer X is a power of 2. Otherwise, return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
return FloorPow2(x) == CeilPow2(x);
Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE instructions and System registers is
// enabled at the target exception level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        boolean disabled;
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPACR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPACR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// ============

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// ================

integer LastActiveElement(bits(N) mask, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return e;
    return -1;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// ================

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED;
```
MaybeZeroSVEUppers()  
// ==-------------------==

MaybeZeroSVEUppers(bits(2) target_el)
  boolean lower_enabled;

  if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
    return;

  if target_el == EL3 then
    if EL2Enabled() then
      lower_enabled = IsFPEnabled(EL2);
    else
      lower_enabled = IsFPEnabled(EL1);
  elsif target_el == EL2 then
    assert !ELUsingAArch32(EL2);
    if HCR_EL2.TGE == '0' then
      lower_enabled = IsFPEnabled(EL1);
    else
      lower_enabled = IsFPEnabled(EL0);
  else
    assert target_el == EL1 && !ELUsingAArch32(EL1);
    lower_enabled = IsFPEnabled(EL0);

  if lower_enabled then
    integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
    integer pl = vl DIV 8;
    for n = 0 to 31
      if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
    for n = 0 to 15
      if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _P[n] = ZeroExtend(_P[n]<pl-1:0>);
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
      _FFR = ZeroExtend(_FFR<pl-1:0>);
// MemNF[] - non-assignment form
// =============================

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccType acctype]
assert size IN {1, 2, 4, 8, 16};
basis(8*size) value;
aligned = (address == Align(address, size));
A = SCTLR[].A;
if !aligned && (A == '1') then
  return (bits(8*size) UNKNOWN, TRUE);
atomic = aligned || size == 1;
if !atomic then
  (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];
  if bad then
    return (bits(8*size) UNKNOWN, TRUE);
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;
  for i = 1 to size-1
    (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];
    if bad then
      return (bits(8*size) UNKNOWN, TRUE);
  else
    (value, bad) = MemSingleNF[address, size, acctype, aligned];
    if bad then
      return (bits(8*size) UNKNOWN, TRUE);
  if BigEndian(acctype) then
    value = BigEndianReverse(value);
return (value, FALSE);
Library pseudocode for aarch64/functions/sve/MemSingleNF

// MemSingleNF[] - non-assignment form
// ===================================

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccType acctype, boolean aligned]
assert acctype IN {AccType_CNOTFIRST, AccType_NONFAULT};
bits(8*size) value;
boolean iswrite = FALSE;
AddressDescriptor memaddrdesc;

// Implementation may suppress NF load for any reason
if ConstrainsUnpredictableBool(Unpredictable_NONFAULT) then
    return (bits(8*size) UNKNOWN, TRUE);

// MMU or MPU
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

// Non-fault load from Device memory must not be performed externally
if memaddrdesc.memattrs.memtype == MemType_Device then
    return (bits(8*size) UNKNOWN, TRUE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (bits(8*size) UNKNOWN, TRUE);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);
if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(address, acctype) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            return (bits(8*size) UNKNOWN, TRUE);

(memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
    if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, size, accdesc) then
        fault = NoFault();
        fault.errortype = memstatus.errortype;
        fault.acctype = memstatus.acctype;
        fault.extflag = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);

return (value, FALSE);

Library pseudocode for aarch64/functions/sve/NoneActive

// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
integer elements = N DIV (esize DIV 8);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
return '1';
Library pseudocode for aarch64/functions/sve/P

// P[] - non-assignment form
// =========================

bits(width) P[integer n]
    assert n >= 0 && n <= 31;
    assert width == PL;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====================

P[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _P[n] = ZeroExtend(value);
    else
        _P[n]<width-1:0> = value;

Library pseudocode for aarch64/functions/sve/PL

// PL - non-assignment form
// ========================

integer PL
    return VL DIV 8;

Library pseudocode for aarch64/functions/sve/PredTest

// PredTest()
// =========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;

Library pseudocode for aarch64/functions/sve/ReducePredicated

// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);    
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;
    for e = 0 to elements-1
        if e * esize < N && ElemP[mask, e, esize] == '1' then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;
    return Reduce(op, operand, esize);
Library pseudocode for aarch64/functions/sve/Reverse

// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
  bits(N) result;
  integer sw = N DIV M;
  assert N == sw * M;
  for s = 0 to sw-1
    Elem[result, (sw - 1) - s, M] = Elem[word, s, M];
  return result;

Library pseudocode for aarch64/functions/sve/SVEAccessTrap

// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
  assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
  route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

  exception = ExceptionSyndrome(Exception_SVEAccessTrap);
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  if route to el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/functions/sve/SVECmp

enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

// Check for 8 bit immediates
if !IsZero(imm<7:0>) then
  // Check for 'ffffffffffffffxy' or '00000000000000xy'
  if !IsZero(imm<63:7>) || !IsOnes(imm<63:7>) then
    return FALSE;

  // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
  if imm<63:32> == imm<31:0> && (!IsZero(imm<31:7>) || !IsOnes(imm<31:7>)) then
    return FALSE;

  // Check for 'ffxyffxyffxyfxy' or '00xy00xy00xy00xy'
    return FALSE;

return TRUE;

Library pseudocode for aarch64/functions/sve/SupportedNonPowerTwoVL

// SupportedNonPowerTwoVL()
// ========================

boolean SupportedNonPowerTwoVL(integer nbits)
return boolean IMPLEMENTATION_DEFINED;

Library pseudocode for aarch64/functions/sve/System

constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
array bits(MAX_VL) _Z[0..31];
array bits(MAX_PL) _P[0..15];
bits(MAX_PL) _FFR;
Library pseudocode for aarch64/functions/sve/VL

```c
// VL - non-assignment form
// ========================

integer VL
    integer vl;
    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);
    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));
    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));
    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);
    return vl;
```

Library pseudocode for aarch64/functions/sve/Z

```c
// Z[] - non-assignment form
// =========================

bits(width) Z[integer n]
    assert n >= 0 && n <= 31;
    assert width == VL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====================

Z[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == VL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

```c
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTKCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```c
type CNTKCTLType;
```
Library pseudocode for aarch64/functions/sysregisters/CPACR

```c
// CPACR[] - non-assignment form
// =============================

CPACRType CPACR[]
    bits(64) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRType

type CPACRType;

Library pseudocode for aarch64/functions/sysregisters/ELR

```c
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1 r = ELR_EL1;
        when EL2 r = ELR_EL2;
        when EL3 r = ELR_EL3;
        otherwise Unreachable();
        return r;
```

```
// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1 ELR_EL1 = r;
        when EL2 ELR_EL2 = r;
        when EL3 ELR_EL3 = r;
        otherwise Unreachable();
        return;
```

Shared Pseudocode Functions  Page 2991
Library pseudocode for aarch64/functions/sysregisters/ESR

// ESR[] - non-assignment form
// ===========================

ESRTypen ESR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = ESR_EL1;
        when EL2 r = ESR_EL2;
        when EL3 r = ESR_EL3;
        otherwise unreachable();
    return r;

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRTypen
    bits(64) r = value;
    case regime of
        when EL1 ESR_EL1 = r;
        when EL2 ESR_EL2 = r;
        when EL3 ESR_EL3 = r;
        otherwise unreachable();
    return;

// ESR[] = ESRTypen
    ESR[SITranslationRegime()] = value;

Library pseudocode for aarch64/functions/sysregisters/ESRTypen

type ESRTypen;
Library pseudocode for aarch64/functions/sysregisters/FAR

// FAR[] - non-assignment form
// ------------------------------------------

bits(64) FAR[bits(2) regime]
   bits(64) r;
   case regime of
      when EL1   r = FAR_EL1;
      when EL2   r = FAR_EL2;
      when EL3   r = FAR_EL3;
      otherwise unreachable();
   return r;

// FAR[] - non-assignment form
// ------------------------------------------

bits(64) FAR[]
   return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// ------------------------

FAR[bits(2) regime] = bits(64) value
   bits(64) r = value;
   case regime of
      when EL1   FAR_EL1 = r;
      when EL2   FAR_EL2 = r;
      when EL3   FAR_EL3 = r;
      otherwise unreachable();
   return;

// FAR[] - assignment form
// ------------------------

FAR[] = bits(64) value
   FAR[S1TranslationRegime()] = value;
   return;

Library pseudocode for aarch64/functions/sysregisters/MAIR

// MAIR[] - non-assignment form
// ------------------------------------------

MAIRType MAIR[bits(2) regime]
   bits(64) r;
   case regime of
      when EL1   r = MAIR_EL1;
      when EL2   r = MAIR_EL2;
      when EL3   r = MAIR_EL3;
      otherwise unreachable();
   return r;

// MAIR[] - non-assignment form
// ------------------------------------------

MAIRType MAIR[]
   return MAIR[S1TranslationRegime()];

Library pseudocode for aarch64/functions/sysregisters/MAIRType

type MAIRType;
SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR<bits(2) regime>
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise unreachable();
    return r;
// SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];

Library pseudocode for aarch64/functions/sysregisters/SCTLR

Library pseudocode for aarch64/functions/sysregisters/SCTLRType
type SCTLRType;

Library pseudocode for aarch64/functions/sysregisters/VBAR
/VBAR[] - non-assignment form
// ============================
bits(64) VBAR<bits(2) regime>
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise unreachable();
    return r;
// VBAR[] - non-assignment form
// ============================
bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

// AArch64.AllocationTagAccessIsEnabled()
// ====================================== 
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(AccType acctype)
    bits(2) el = AArch64.AccessUsesEL(acctype);
    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '1' then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && el == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && el == EL0 && !((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11')) then
        return FALSE;
    else
        return TRUE;

Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

// AArch64.ChooseNonExcludedTag()
// ============================== 
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;
    if IsOnes(exclude) then
        return '0000';
    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';
        while offset != '0000' do
            offset = offset - '0001';
            tag = tag + '0001';
        return tag;

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

// AArch64.ExecutingBROrBLROrRetInstr()
// ====================================
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    instr = ThisInstr();
    if !HaveBTIExt() then return FALSE;
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

// AArch64.ExecutingBTIInstr()
// ===========================
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
if !HaveBTIExt() then return FALSE;

instr = ThisInstr();
if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
  CRm  = instr<11:8>;
  op2  = instr<7:5>;
  return (CRm == '0100' && op2<0> == '0');
else
  return FALSE;

Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

// AArch64.ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
instr = ThisInstr();
return instr<31:12> == '11010110100111110000';

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr

// Execute an implementation-defined system instruction with write (source operand).
AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstrWithResult

// Execute an implementation-defined system instruction with read (result operand).
AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2);

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegRead

// Read from an implementation-defined system register and write the contents of the register to X[t].
AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite

// Write to an implementation-defined system register.
AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);

Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

// AArch64.NextRandomTagBit()
// ==========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
bits(16) lfsr = RGSR_EL1.SEED;
bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
RGSR_EL1.SEED = top:lfsr<15:1>;
return top;
Library pseudocode for aarch64/functions/system/AArch64.RandomTag

// AArch64.RandomTag()
// ===============
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
bits(4) tag;
for i = 0 to 3
  tag<i> = AArch64.NextRandomTagBit();
return tag;

Library pseudocode for aarch64/functions/system/AArch64.SysInstr

// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].
AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

// Read from a system register and write the contents of the register to X[t].
AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);

Library pseudocode for aarch64/functions/system/BTypeCompatible

boolean BTypeCompatible;

Library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

// BTypeCompatible_BTI
// ===============
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
case hintcode of
  when '00'
    return FALSE;
  when '01'
    return PSTATE.BTYPE != '11';
  when '10'
    return PSTATE.BTYPE != '10';
  when '11'
    return TRUE;
Library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

// BTypeCompatible_PACIXSP()
// =========================
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == EL0
            then 35 else 36;
        return SCTLR[index] == '0';
    else
        return FALSE;

Library pseudocode for aarch64/functions/system/BTypeNext

bits(2) BTypeNext;

Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
// //
// // The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// // exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
// // This function is permitted to generate a non-deterministic selection from the set of non-excluded
// // Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// // GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// // may choose to behave the same as GCR_EL1.RRND=0.
bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);

Library pseudocode for aarch64/functions/system/InGuardedPage

boolean InGuardedPage;

Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

// IsHCRXEL2Enabled()
// ==================
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    assert(HaveFeatHCX());
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;
    return EL2Enabled();

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
Library pseudocode for aarch64/functions/system/SetBTypeNext

```c
// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI
SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

Library pseudocode for aarch64/functions/system/SetInGuardedPage

```c
// SetInGuardedPage()
// ==================
// Global state updated to denote if memory access is from a guarded page.
SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

Library pseudocode for aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```c
// AArch64.ExceptionReturn()
// =========================
AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if HaveIESB() then
        sync_errors = SCTLR[].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
            SynchronizeContext();
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();
    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
        else                                     // Return to AArch64
            // ELR_ELx[63:56] might include a tag
            new_pc = AArch64.BranchAddr(new_pc);
    if UsingAArch32() then
        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);
    CheckExceptionCatch(FALSE);              // Check for debug event on exception return
```
Library pseudocode for aarch64/instrs/countop/CountOp

```c
```

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

```c
// DecodeRegExtend()  
// ===============  
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
    when '000' return ExtendType_UXTB;
    when '001' return ExtendType_UXTH;
    when '010' return ExtendType_UXTW;
    when '011' return ExtendType_UXTX;
    when '100' return ExtendType_SXTB;
    when '101' return ExtendType_SXTH;
    when '110' return ExtendType_SXTW;
    when '111' return ExtendType_SXTX;
```

Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```c
// ExtendReg()
// ============
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;
    case exttype of
    when ExtendType_SXTB unsigned = FALSE; len = 8;
    when ExtendType_SXTH unsigned = FALSE; len = 16;
    when ExtendType_SXTW unsigned = FALSE; len = 32;
    when ExtendType_SXTX unsigned = FALSE; len = 64;
    when ExtendType_UXTB unsigned = TRUE; len = 8;
    when ExtendType_UXTH unsigned = TRUE; len = 16;
    when ExtendType_UXTW unsigned = TRUE; len = 32;
    when ExtendType_UXTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);
    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

Library pseudocode for aarch64/instrs/extendreg/ExtendType

```c
enumeration ExtendType {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                        ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```c
enumeration FPMaxMinOp {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                        FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```
Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp   {
    FPUnaryOp_ABS, FPUnaryOp_MOV,
    FPUnaryOp_NEG, FPUnaryOp_SQRT;
}
```

Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {
    FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
    FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
};
```

Library pseudocode for aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;
    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;
    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;
    // must be UBFX/SBFX alias
    return TRUE;
```
// DecodeBitMasks()
// ================

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)

  bits(64) tmask, wmask;
  bits(6) tmask_and, wmask_and;
  bits(6) tmask_or, wmask_or;
  bits(6) levels;

  // Compute log2 of element size
  // 2^len must be in range [2, M]
  len = HighestSetBit(immN:NOT(imms));
  if len < 1 then UNDEFINED;
  assert M >= (1 << len);

  // Determine S, R and S - R parameters
  levels = ZeroExtend(Ones(len), 6);

  // For logical immediates an all-ones value of S is reserved
  // since it would generate a useless all-ones result (many times)
  if immediate && (imms AND levels) == levels then
    UNDEFINED;
  S = UInt(imms AND levels);
  R = UInt(immr AND levels);
  diff = S - R;    // 6-bit subtract with borrow

  // From a software perspective, the remaining code is equivalent to:
  //   esize = 1 << len;
  //   d = UInt(diff<len-1:0>);
  //   welem = ZeroExtend(Ones(S + 1), esize);
  //   telem = ZeroExtend(Ones(d + 1), esize);
  //   wmask = Replicate(ROR(welem, R));
  //   tmask = Replicate(telem);
  //   return (wmask, tmask);

  // Compute "top mask"
  tmask_and = diff<5:0> OR NOT(levels);
  tmask_or  = diff<5:0> AND levels;

  tmask = Ones(64);
  tmask = ((tmask
      AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
    OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));

  // optimization of first step:
  // tmask = Replicate(tmask_and<0> : '1', 32);
  tmask = ((tmask
      AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
    OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
  tmask = ((tmask
      AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
    OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
  tmask = ((tmask
      AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
    OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
  tmask = ((tmask
      AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
    OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
  tmask = ((tmask
      AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
    OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

  // Compute "wraparound mask"
  wmask_and = immr OR NOT(levels);
  wmask_or  = immr AND levels;

  wmask = Zeros(64);
  wmask = ((wmask
AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));

// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);

wmask = ((wmask
AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));

wmask = ((wmask
AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));

wmask = ((wmask
AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));

wmask = ((wmask
AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));

wmask = ((wmask
AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from S - R
  wmask = wmask AND tmask;
else
  wmask = wmask OR tmask;
return (wmask<M-1:0>, tmask<M-1:0>);

### Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/
#### MoveWideOp

```cpp
```

### Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```cpp
// MoveWidePreferred()
// ---------------------
// // Return TRUE if a bitmask immediate encoding would generate an immediate
// // value that could also be represented by a single MOVZ or MOVN instruction.
// // Used as a condition for the preferred MOV<->ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
{
  integer S = UInt(imms);
  integer R = UInt(immr);
  integer width = if sf == '1' then 64 else 32;

  // element size must equal total immediate size
  if sf == '1' && immN:imms != '1xxxxxx' then
    return FALSE;
  if sf == '0' && immN:imms != '00xxxxx' then
    return FALSE;

  // for MOVZ must contain no more than 16 ones
  if S < 16 then
    // ones must not span halfword boundary when rotated
    return (-R MOD 16) <= (15 - S);

  // for MOVN must contain no more than 16 zeros
  if S >= width - 15 then
    // zeros must not span halfword boundary when rotated
    return (R MOD 16) <= (S - (width - 15));

  return FALSE;
}
```
Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

// DecodeShift()
// =============
// Decode shift encodings

SHIFTTYPE DecodeShift(bits(2) op)
  case op of
    when '00' return ShiftType_LSL;
    when '01' return ShiftType_LSR;
    when '10' return ShiftType_ASR;
    when '11' return ShiftType_ROR;

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

// ShiftReg()
// =========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, SHIFTTYPE shiftype, integer amount)
  bits(N) result = [reg];
  case shiftype of
    when ShiftType_LSL result = LSL(result, amount);
    when ShiftType_LSR result = LSR(result, amount);
    when ShiftType_ASR result = ASR(result, amount);
    when ShiftType_ROR result = ROR(result, amount);
  return result;

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

enumeration MemAtomicOp {MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP};

Library pseudocode for aarch64/instrs/memory/memop/MemOp

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

// Prefetch()
// =========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop) {
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;  // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;  // PLI: preload instructions
        when '10' hint = Prefetch_WRITE; // PST: prepare for store
        when '11' return;               // unallocated hint
    target = UInt(prfop<2:1>);       // target cache level
    stream = (prfop<0> != '0');       // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
}

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

enumeration MemBarrierOp {
    MemBarrierOp_DSB         // Data Synchronization Barrier
    , MemBarrierOp_DMB        // Data Memory Barrier
    , MemBarrierOp_ISB        // Instruction Synchronization Barrier
    , MemBarrierOp_SSBB       // Speculative Synchronization Barrier to VA
    , MemBarrierOp_PSSBB      // Speculative Synchronization Barrier to PA
    , MemBarrierOp_SB         // Speculation Barrier
};

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CSDB
};

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

enumeration PSTATEField {
    PSTATEField_DAIFSet, PSTATEField_DAIFClr,
    PSTATEField_PAN,   // Armv8.1
    PSTATEField_UA0,   // Armv8.2
    PSTATEField_DIT,   // Armv8.4
    PSTATEField_SSSB,
    PSTATEField_TCO,   // Armv8.5
    PSTATEField_ALLINT,
    PSTATEField_SP
};
AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess)

TranslationStage stage = stage_in;
bits(2) el = el_in;

// For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
// the EL2&0 translation regime is used.
if HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage_1 then
    el = EL2;
if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() then
    stage = TranslationStage_1;

acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;
iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
aligned = TRUE;
ispriv = el != EL0;

fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

Regime regime;
if stage == TranslationStage_12 then
    regime = Regime_EL10;
else
    regime = TranslationRegime(el, acctype);

AddressDescriptor addrdesc;
ss = SecurityStateAtEL(el);
if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, address<31:0>, acctype,
                                                  aligned, iswrite, ispriv);
    else
        (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, ss, address<31:0>, acctype,
                                                  aligned, iswrite, ispriv);
else
    (fault, addrdesc) = AArch64.S1Translate(fault, regime, ss, address, acctype, aligned, iswrite, ispriv);

if stage == TranslationStage_12 && fault.statuscode == Fault_None then
    if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
        addrdesc.vaddress = ZeroExtend(address);
        s2fs1walk = FALSE;
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fs1walk, acctype,
                                                aligned, iswrite, ispriv);
    elsif regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        s2fs1walk = FALSE;
        (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, s1aarch64, ss, s2fs1walk,
                                                 acctype, aligned, iswrite, ispriv);
else
    is_ATS1Ex = stage != TranslationStage_12;
if fault.statuscode != Fault_None then
    addrdesc = CreateFaultyAddressDescriptor(address, fault);
    // Take an exception on:
    // * A Synchronous external abort occurs on translation table walk
    // * A stage 2 fault occurs on a stage 1 walk
    if IsExternalAbort(fault) || (PSTATE.EL == EL1 && fault.s2fs1walk) then
        PAR_EL1 = bits(64) UNKNOWN;
        AArch64.Abort(address, addrdesc.fault);
        AArch64.EncodePAR(regime, addrdesc);
    return;
// AArch64.EncodePAR()
// ================
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, AddressDescriptor addrdesc)
    PAR_EL1 = Zeros();
    paspace = addrdesc.paddress.paspace;

    if IsFault(addrdesc) then
        PAR_EL1.F = '0';
        PAR_EL1<11> = '1'; // RES1
        if SecurityStateForRegime(regime) == SS_Secure then
            PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
        else
            PAR_EL1.NS = bit UNKNOWN;
        end
        PAR_EL1.SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR_EL1.PA   = addrdesc.paddress.address<52-1:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F = '1';
        PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fswalk then '1' else '0';
        PAR_EL1.S   = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    end
    return;

// AArch64.PARFaultStatus()
// =========================
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    end
    return fst;
Library pseudocode for aarch64/instrs/system/sysops/dc/AArch64.DC
// AArch64.DC()
// ============
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
AArch64.DC

CacheOpScope opscope = opscope_in;
AccType acctype = AccType_DC;
CacheRecord cache;

  cache.acctype = acctype;
  cache.cachetype = cachetype;
  cache.cacheop = cacheop;
  cache.opscope = opscope;

if opscope == CacheOpScope_SetWay then
  ss = SecurityStateAtEL(PSTATE.EL);
  cache.cpas = CPASAtSecurityState(ss);
  cache.shareability = Shareability_NS;
  (cache.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
    (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
  cache.cacheop = CacheOp_CleanInvalidate;
  CACHE_OP(cache);
  return;
if EL2Enabled() && !IsInHost() then
  if PSTATE.EL IN {EL0, EL1} then
    cache.is_vmid_valid = TRUE;
    cache.vmid = VMID[];
  else
    cache.is_vmid_valid = FALSE;
else
  cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
  cache.is_asid_valid = TRUE;
  cache.asid = ASID[];
else
  cache.is_asid_valid = FALSE;
if opscope == CacheOpScope_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP" then
  opscope = CacheOpScope_PoP;
if opscope == CacheOpScope_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP" then
  opscope = CacheOpScope_PoC;
need_translate = DCInstNeedsTranslation(opscope);
iswrite = cacheop == CacheOp_Invalidate;
vaddress = regval;
size = 0; // by default no watchpoint address
if iswrite then
  size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
  assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
  assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
  vaddr = Align(regval, size);
  cache.translated = need_translate;
  cache.vaddress = vaddress;
if need_translate then
  wasaligned = TRUE;
  memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
  if IsFault(memaddrdesc) then
    AArch64.Abort(regval, memaddrdesc.fault);
  memattrs = memaddrdesc.memattrs;
  cache.paddress = memaddrdesc.paddress;
  cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
  cache.shareability = memattrs.shareability;
else
  cache.shareability = Shareability_NSH;
else
  cache.shareability = Shareability UNKNOWN;
  cache.paddress = FullAddress UNKNOWN;

if cacheop == CacheOp_Invalidate & PSTATE.EL == EL1 & EL2Enabled() & HCR_EL2.<DC,VM> != '00' then
  cache.cacheop = CacheOp_CleanInvalidate;
  CACHE_OP(cache);
return;

Library pseudocode for aarch64/intrs/system/sysops/dc/AArch64.MemZero

// AArch64.MemZero()
// =================
AArch64.MemZero(bits(64) regval, CacheType cachetype)

  AccType acctype = AccType_DCZVA;
  boolean iswrite = TRUE;
  boolean wasaligned = TRUE;

  integer size = 4*(2^(UInt(DCZID_EL0.BS)));
  bits(64) vaddress = Align(regval, size);

  memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

  if IsFault(memaddrdesc) then
    if IsDebugException(memaddrdesc.fault) then
      AArch64.Abort(vaddress, memaddrdesc.fault);
    else
      AArch64.Abort(regval, memaddrdesc.fault);
  else
    if cachetype == CacheType_Data then
      AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
    elsif cachetype == CacheType_Tag then
      if HaveMTEExt() then
        AArch64.TagMemZero(vaddress, size);
      end if;
    elsif cachetype == CacheType_Data_Tag then
      if HaveMTEExt() then
        AArch64.TagMemZero(vaddress, size);
      end if;
      AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
    return;
// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(
CacheOpScope opscope
) regval = bits(64) UNKNOWN;
AArch64.IC(regval, opscope);

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
CacheRecord cache;
AccType accctype = AccType_IC;

    cache.acctype = accctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if (opscope == CacheOpScope_ALLUIS ||
            (opscope == CacheOpScope_ALLU &&
             PSTATE.EL == EL1 &&
             EL2Enabled() &&
             HCR_EL2.FB == '1')) then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
            cache.is_asid_valid = TRUE;
            cache.asid = ASID[];
        else
            cache.is_vmid_valid = FALSE;
            if PSTATE.EL == EL0 then
                cache.is_asid_valid = TRUE;
            else
                cache.is_asid_valid = FALSE;
            bits(64) vaddress = regval;
            need_translate = ICInstNeedsTranslation(opscope);

            cache.vaddress = regval;
            cache.shareability = Shareability_NSH;
            cache.translated = need_translate;

            if !need_translate then
                cache.paddress = FullAddress UNKNOWN;
                CACHE_OP(cache);
            return;
            iswrite = FALSE;
            wasaligned = TRUE;
            size = 0;
            memaddrdesc = AArch64.TranslateAddress(vaddress, accctype, iswrite, wasaligned, size);

            if IsFault(memaddrdesc) then
                AArch64.Abort(regval, memaddrdesc.fault);

            cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
            cache.paddress = memaddrdesc.paddress;
        else
            // Shared Pseudocode Functions
            Page 3013
CACHE_OP(cache);
return;

Library pseudocode for aarch64/instrs/system/sysops/predictionrestrict/RestrictPrediction

// RestrictPrediction()
// ====================
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
target_el = val<25:24>;

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if UInt(target_el) > UInt(PSTATE.EL) then return;

    bit ns = val<26>;
    ss = TargetSecurityState(ns);

    c.security = ss;
c.target_el = target_el;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
c.all_vmid = FALSE;
c.vmid = VMID[
        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
c.all_vmid = val<48> == '1';
c.vmid = val<47:32>;
        // Only valid if  val<48> == '0';
    else
        c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
c.all_asid = FALSE;
c.asid = ASID[
    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
c.all_asid = val<16> == '1';
c.asid = val<15:0>;
        // Only valid if  val<16> == '0';
    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
// SysOp()
// =======
SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2) of
  case op1:CRn:CRm:op2 of
    when '000 0111 1000 000' return Sys_AT; // S1E1R
    when '100 0111 1000 000' return Sys_AT; // S1E2R
    when '110 0111 1000 000' return Sys_AT; // S1E3R
    when '000 0111 1000 001' return Sys_AT; // S1E1W
    when '100 0111 1000 001' return Sys_AT; // S1E2W
    when '110 0111 1000 001' return Sys_AT; // S1E3W
    when '000 0111 1000 010' return Sys_AT; // S1E0R
    when '000 0111 1000 011' return Sys_AT; // S1E0W
    when '100 0111 1000 100' return Sys_AT; // S12E1R
    when '100 0111 1000 101' return Sys_AT; // S12E1W
    when '100 0111 1000 110' return Sys_AT; // S12E0R
    when '100 0111 1000 111' return Sys_AT; // S12E0W
    when '011 0111 0100 001' return Sys_DC; // ZVA
    when '000 0111 0110 001' return Sys_DC; // IVAC
    when '000 0111 0110 010' return Sys_DC; // ISW
    when '011 0111 1010 001' return Sys_DC; // CVAC
    when '000 0111 1010 010' return Sys_DC; // CSW
    when '011 0111 1011 001' return Sys_DC; // CVAU
    when '011 0111 1110 001' return Sys_DC; // CIVAC
    when '000 0111 1110 010' return Sys_DC; // CISW
    when '011 0111 1101 001' return Sys_DC; // CVADP
    when '000 0111 0001 000' return Sys_IC; // IALLUIS
    when '000 0111 0101 000' return Sys_IC; // IALLU
    when '011 0111 0101 001' return Sys_IC; // IVAU
    when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
    when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
    when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
    when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
    when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
    when '000 1000 0011 001' return Sys_TLBI; // VAEIIS
    when '110 1000 0011 001' return Sys_TLBI; // VAE2IS
    when '000 1000 0011 011' return Sys_TLBI; // VAE3IS
    when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
    when '100 1000 0011 100' return Sys_TLBI; // VMALE2IS
    when '100 1000 0011 101' return Sys_TLBI; // VMALE3IS
    when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
    when '100 1000 0011 111' return Sys_TLBI; // VAALE1IS
    when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
    when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
    when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
    when '100 1000 0111 000' return Sys_TLBI; // ALLE2IS
    when '110 1000 0111 000' return Sys_TLBI; // ALLE3IS
    when '000 1000 0111 001' return Sys_TLBI; // VAEIIS
    when '110 1000 0111 001' return Sys_TLBI; // VAE2IS
    when '000 1000 0111 011' return Sys_TLBI; // VAE3IS
    when '000 1000 0111 000' return Sys_TLBI; // VMALLE1IS
    when '100 1000 0111 100' return Sys_TLBI; // VMALE2IS
    when '100 1000 0111 101' return Sys_TLBI; // VMALE3IS
    when '110 1000 0111 100' return Sys_TLBI; // VMALLS12E1IS
    when '110 1000 0111 110' return Sys_TLBI; // VAALE1IS
    when '000 1000 0111 000' return Sys_TLBI; // VMALEIS
    when '100 1000 0111 000' return Sys_TLBI; // VALE2IS
    when '100 1000 0111 001' return Sys_TLBI; // VALE3IS
    when '000 1000 0111 001' return Sys_TLBI; // VAAEIS
    when '100 1000 0111 011' return Sys_TLBI; // VAAE2IS
    when '100 1000 0111 101' return Sys_TLBI; // VVAE1IS
    when '100 1000 0111 111' return Sys_TLBI; // VVAE2IS

  return Sys_SYS;

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
// AArch32.DTLBI_ALL()
// ==============
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN (EL3, EL2, EL1);

TLBIRecord r;
r.op = TLBIOp_DALL;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.level = TLBILevel_Any;
r.attr = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

// AArch32.DTLBI_ASID()
// ===========
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN (EL3, EL2, EL1);

TLBIRecord r;
r.op = TLBIOp_DASID;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = TLBILevel_Any;
r.attr = attr;
r.asid = Zeros(8) : Rt<7:0>;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_VA

// AArch32.DTLBI_VA()
// ===============
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_DVA;
array_from_aarch64 = FALSE;
security = security;
regime = regime;
vmid = vmid;
level = level;
attr = attr;
asid = Zeros(8) : Rt<7:0>;
address = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ALL

// AArch32.ITLBI_ALL()
// ================
// Invalidate all instruction TLB entries for the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_IALL;
array_from_aarch64 = FALSE;
security = security;
regime = regime;
level = TLBILevel_Any;
attr = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch32.ITLBI_ASID()
// ===============
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBIRecord r;
r.op = TLBIOp_IASID;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = TLBILevel_Any;
r.attr = attr;
r.asid = Zeros(8) : Rt<7:0>;
TLB(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

// AArch32.ITLBI_VA()
// ===============
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBIRecord r;
r.op = TLBIOp_IVA;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.asid = Zeros(8) : Rt<7:0>;
r.address = Zeros(32) : Rt<31:12> : Zeros(12);
TLB(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
r.op           = TLBIOp_ALL;
r.from_aarch64 = FALSE;
r.security     = security;
r.regime       = regime;
r.level        = TLBILevel_Any;
r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op           = TLBIOp_ASID;
r.from_aarch64 = FALSE;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = TLBILevel_Any;
r.attr         = attr;
r.asid         = Zeros(8) : Rt<7:0>;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch32.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2};
assert security == SS_NonSecure;

TLBIRecord r;
r.op = TLBIOp_IPAS2;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.address = Zeros(24) : Rt<27:0> : Zeros(12);
r.ipaspace = PAS_NonSecure;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch32.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    TLBIRecord r;
    r.op = TLBIOp_VA;
    r.from_aarch64 = FALSE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Zeros(8) : Rt<7:0>;
    r.address = Zeros(32) : Rt<31:12> : Zeros(12);
    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
// AArch32.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBI0p_VAA;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;
r.address = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

// AArch32.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 2 only entries are not in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBI0p_VMALL;
r.from_aarch64 = FALSE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = TLBILevel_Any;
r.attr = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch32.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
      r.op           = TLBIOp_VMALLS12;
      r.from_aarch64 = FALSE;
      r.security     = security;
      r.regime       = regime;
      r.level        = TLBILevel_Any;
      r.vmid         = vmid;
      r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

// AArch64.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidiation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
      r.op           = TLBIOp_ALL;
      r.from_aarch64 = TRUE;
      r.security     = security;
      r.regime       = regime;
      r.level        = TLBILevel_Any;
      r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch64.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBRecord r;
r.op = TLBIOp_ASID;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = TLBILevel_Any;
r.attr = attr;
r.asid = Xt<63:48>;

TLBI(r);
if shareability != Shareability_NS then Broadcast(shareability, r);
return;
// AArch64.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2};

TLBIR e;   
    r.op = TLBIOp_IPAS2; 
    r.from_aarch64 = TRUE; 
    r.security = security; 
    r.regime = regime; 
    r.vmid = vmid; 
    r.level = level; 
    r.attr = attr; 
    r.address = ZeroExtend(Xt<39:0> : Zeros(12));

    case security of
       when SS_NonSecure
           r.ipaspace = PAS_NonSecure;
       when SS_Secure
           r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch64.TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
// TLBLevel_Any : this applies to TLB entries at all levels
// TLBLevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
{r.op = TLBIOp_RIPAS2;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;

bits(2) tg = Xt<47:46>;
integer scale = UInt(Xt<45:44>);
integer num = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);
boolean valid;
(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);
if !valid then return;

case security of
  when SS_NonSecure
    r.ipaspace = PAS_NonSecure;
  when SS_Secure
    r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
AArch64.TLBI_RVA(
    SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.asid = Xt<63:48>;

boolean valid;

(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
// AArch64.TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//   TLBILevel_Any : this applies to TLB entries at all levels
//   TLBILevel_Last: this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed. 
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid, 
Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op = TLBIOp_RVAA;
r.from_aarch64 = TRUE;
r.security = security;
r.regime = regime;
r.vmid = vmid;
r.level = level;
r.attr = attr;

bits(2) tg = Xt<47:46>;
integer scale = UInt(Xt<45:44>);
integer num = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

boolean valid;

(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability_NS then Broadcast(shareability, r);
return;
Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VA

// AArch64.TLBI_VA()
// =============
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
// TLBILevel_Any : this applies to TLB entries at all levels
// TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12));

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
A pseudo-code for AArch64.TLBI_VAA

```plaintext
AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBIRecord r;
    r.op = TLBI0p_VAA;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.address = ZeroExtend(Xt<43:0> : Zeros(12));
TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
```

A pseudo-code for AArch64.TLBI_VMALL

```plaintext
AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability, TLBILevel level, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};
TLBIRecord r;
    r.op = TLBI0p_VMALL;
    r.from_aarch64 = TRUE;
    r.security = security;
    r.regime = regime;
    r.level = TLBILevel_Any;
    r.vmid = vmid;
    r.attr = attr;
TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;
```
// AArch64.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
r.op           = TLBIOp_VMALLS12;
r.from_aarch64 = TRUE;
r.security     = security;
r.regime       = regime;
r.level        = TLBILevel_Any;
r.vmid         = vmid;
r.attr         = attr;

TLBI(r);
if shareability != Shareability_NS then Broadcast(shareability, r);
return;

constant bits(16) ASID_NONE = Zeros();
// HasLargeAddress()
// ================
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
if !Have52BitIPAAndPASpaceExt() then
    return FALSE;
case regime of
    when Regime_EL3
        return TCR_EL3<32> == '1';
    when Regime_EL2
        return TCR_EL2<32> == '1';
    when Regime_EL20
        return TCR_EL2<59> == '1';
    when Regime_EL10
        return TCR_EL1<59> == '1';
    otherwise
        Unreachable();

// TLBI()
// ======
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.

TLBI(TLBIRecord r)
IMPLEMENTATION_DEFINED;

enumeration TLBILevel {
    TLBILevel_Any,
    TLBILevel_Last
};
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIMatch
// TLBIMatch()
// ===========
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBIRecord entry)
boolean match;
case tlbi.op of
  when TLBIOp_DALL, TLBIOp_TALL
    match = (tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime);
  when TLBIOp_DASID, TLBIOp_IASID
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              (UseASID(entry.context) && entry.context.nG == '1' &&
               tlbi.asid == entry.context.asid));
  when TLBIOp_DVA, TLBIOp_IVA
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              (UseASID(entry.context) && entry.context.nG == '1' &&
               tlbi.asid == entry.context.asid) ||
              (tlbi.level == TLBILevel_Any | | !entry.walkstate.istable));
  when TLBIOp_ALL
    relax_regime = (tlbi.from_aarch64 &&
                    tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                    entry.context.regime IN {Regime_EL20, Regime_EL2});
    match = (tlbi.security == entry.context.ss &&
             (tlbi.regime == entry.context.regime | | relax_regime));
  when TLBIOp_ASID
    match = (entry.context.includes_s1 &&
              entry.context.includes_s2 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              (UseASID(entry.context) && entry.context.nG == '1' &&
               tlbi.asid == entry.context.asid));
  when TLBIOp_IPAS2
    match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              (UseASID(entry.context) && entry.context.nG == '0' &&
               tlbi.ipaspace == entry.context.ipaspace) &&
              tlbi.address<51:entry.blocksize> == entry.context.ia<51:entry.blocksize> &&
              (tlbi.level == TLBILevel_Any | | !entry.walkstate.istable));
  when TLBIOp_VAA
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
              tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
              (tlbi.level == TLBILevel_Any | | !entry.walkstate.istable));
  when TLBIOp_VMALL
    match = (entry.context.includes_s1 &&
              tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
  when TLBIOp_VMALLS12
    match = (tlbi.security == entry.context.ss &&
              tlbi.regime   == entry.context.regime &&
              (UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));

// Shared Pseudocode Functions
```plaintext
tlbi.regime == entry.context.regime &&
(!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
when TLBIOp_RIPAS2
  match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
    tlbi.security == entry.context.ss &&
   tlbi.regime == entry.context.regime &&
   (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
   tlbi.ipaspace == entry.context.ipaspace &&
   (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
   UInt(tlbi.address) <= UInt(entry.context.ia) &&
   UInt(tlbi.end_address) > UInt(entry.context.ia));
when TLBIOp_RVAA
  match = (entry.context.includes_s1 &&
    tlbi.security == entry.context.ss &&
   tlbi.regime == entry.context.regime &&
   (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
   (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
   UInt(tlbi.address) <= UInt(entry.context.ia) &&
   UInt(tlbi.end_address) > UInt(entry.context.ia));
when TLBIOp_RVA
  match = (entry.context.includes_s1 &&
    tlbi.security == entry.context.ss &&
   tlbi.regime == entry.context.regime &&
   (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
   (!UseASID(entry.context) || tlbi.asid == entry.context.asid || entry.context.nG == '0') &&
   (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
   UInt(tlbi.address) <= UInt(entry.context.ia) &&
   UInt(tlbi.end_address) > UInt(entry.context.ia));
if tlbi.attr == TLBI_ExcludeXS && entry.context.xs == '1' then
  match = FALSE;
return match;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIMemAttr

enumeration TLBIMemAttr {
  TLBI_AllAttr,
  TLBI_ExcludeXS
};

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIOp

enumeration TLBIOp {
  TLBIOp_DALL,       // AArch32 Data TLBI operations - deprecated
  TLBIOp_DASID,
  TLBIOp_DVA,
  TLBIOp_IALL,       // AArch32 Instruction TLBI operations - deprecated
  TLBIOp_IASID,
  TLBIOp_IVA,
  TLBIOp_ALL,
  TLBIOp_IPAS2,
  TLBIOp_VAA,
  TLBIOp_VA,
  TLBIOp_VMALL,
  TLBIOp_VMALLS12,
  TLBIOp_RIPAS2,
  TLBIOp_RVAA,
  TLBIOp_RVA,
};
```
// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)

  boolean valid = TRUE;
  bits(64) start = Zeros(64);
  bits(64) end   = Zeros(64);

  bits(2) tg        = Xt<47:46>;
  integer scale     = UInt(Xt<45:44>);
  integer num       = UInt(Xt<43:39>);
  integer tg_bits;
  if tg == '00' then
    return (FALSE, tg, start, end);
  case tg of
    when '01' // 4KB
      tg_bits = 12;
      if HasLargeAddress(regime) then
        start<52:16> = Xt<36:0>;
        start<63:53> = Replicate(Xt<36>, 11);
      else
        start<48:12> = Xt<36:0>;
        start<63:49> = Replicate(Xt<36>, 15);
    when '10' // 16KB
      tg_bits = 14;
      if HasLargeAddress(regime) then
        start<52:16> = Xt<36:0>;
        start<63:53> = Replicate(Xt<36>, 11);
      else
        start<50:14> = Xt<36:0>;
        start<63:51> = Replicate(Xt<36>, 13);
    when '11' // 64KB
      tg_bits = 16;
      start<52:16> = Xt<36:0>;
      start<63:53> = Replicate(Xt<36>, 11);
    otherwise
      Unreachable();
  end
t
  integer range = (num+1) << (5*scale + 1 + tg_bits);
  end   = start + range<63:0>;

  if end<52> ! = start<52> then
    // overflow, saturate it
    end = Replicate(start<52>, 64-52) : Ones(52);

  return (valid, tg, start, end);

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRange

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRange

type TLBIRange is ( TLBIOp op, SecurityState security, Regime regime, bits(16) vmid, bits(16) asid, TLBILevel level, TLBIMemAttr attr, PASpace ipaspace, bits(64) address, bits(64) end_address, bits(2) tg )
// VMID[]
// ======
// Effective VMID.

bits(16) VMID[]
  if EL2Enabled() then
    if !ELUsingAArch32(EL2) then
      if Have16bitVMID() && VTCR_EL2.VS == '1' then
        return VTTBR_EL2.VMID;
      else
        return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
      return ZeroExtend(VTTBR.VMID, 16);
  elsif HaveEL(EL2) && HaveSecureEL2Ext() then
    return Zeros(16);
  else
    return VMID_NONE;

constant bits(16) VMID_NONE = Zeros();

VBitOp = enumeration
  VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR;

CompareOp = enumeration

ImmediateOp = enumeration
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// ========

// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    if N == esize then
        return input<esize-1:0>;
    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);
    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;
        return result;

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                        ReduceOp_FMIN, ReduceOp_FMAX,
                        ReduceOp_FADD, ReduceOp_ADD};
Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype_in, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
assert (UsingAArch32() && size IN {2,4}) || size == 4;
AccType acctype = acctype_in;
match = FALSE;
for i = 0 to NumBreakpointsImplemented() - 1
  match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
  match = match || match_i;
if match && HaltOnBreakpointOrWatchpoint() then
  reason = DebugHalt_Breakpoint;
  Halt(reason);
elsif match then
  acctype = AccType_IFETCH;
  iswrite = FALSE;
  return AArch64.DebugFault(acctype, iswrite);
else
  return NoFault();

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
FaultRecord fault = NoFault();
boolean generate_exception;
d_side = (acctype != AccType_IFETCH);
if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
  mask = '0';
  generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, IsSecure(), mask) && MDSCR_EL1.MDE == '1';
else
  generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
halt = HaltOnBreakpointOrWatchpoint();
if generate_exception || halt then
  if d_side then
    fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
  else
    fault = AArch64.CheckBreakpoint(vaddress, acctype, size);
  return fault;
library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype, boolean iswrite_in, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
boolean iswrite = iswrite_in;

if acctype IN {AccType_TTW, AccType_IC, AccType_AT, AccType_ATPAN} then
    return NoFault();
if acctype == AccType_DC then
    if !iswrite then
        return NoFault();
match = FALSE;
match_on_read = FALSE;
ispriv = AArch64.AccessUsesEL(acctype) != EL0;
for i = 0 to NumWatchpointsImplemented() - 1
    if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
        match = TRUE;
        if DBGWCR_EL1[i].LSC<0> == '1' then
            match_on_read = TRUE;

if match && acctype == AccType_ATOMICRW then
    iswrite = !match_on_read;
if match && HaltonBreakpointOrWatchpoint() then
    if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
        reason = DebugHalt_Watchpoint;
        EDWAR = vaddress;
        Halt(reason);
else
    // Fault will be reported and cancelled
    return AArch64.DebugFault(acctype, iswrite);
elsif match then
    return AArch64.DebugFault(acctype, iswrite);
else
    return NoFault();
Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.BlockBase

// AArch64.BlockBase()
// ====================
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block

bits(52) AArch64.BlockBase(bits(64) descriptor, bit ds, TGx tgx, integer level)
bits(52) blockbase = Zeros();
if tgx == TGx_4KB && level == 2 then
    blockbase<47:21> = descriptor<47:21>;
elsif tgx == TGx_4KB && level == 1 then
    blockbase<47:30> = descriptor<47:30>;
elsif tgx == TGx_4KB && level == 0 then
    blockbase<47:39> = descriptor<47:39>;
elsif tgx == TGx_16KB && level == 2 then
    blockbase<47:25> = descriptor<47:25>;
elsif tgx == TGx_16KB && level == 1 then
    blockbase<47:36> = descriptor<47:36>;
elsif tgx == TGx_64KB && level == 2 then
    blockbase<47:29> = descriptor<47:29>;
elsif tgx == TGx_64KB && level == 1 then
    blockbase<47:42> = descriptor<47:42>;
else
    Unreachable();
if Have52BitPAExt() && tgx == TGx_64KB then
    blockbase<51:12> = descriptor<15:12>;
elsif ds == '1' then
    blockbase<51:48> = descriptor<9:8>:descriptor<49:48>;
return blockbase;

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.IASize

// AArch64.IASize()
// ================
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
return 64 - UInt(txsz);

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.NextTableBase

// AArch64.NextTableBase()
// =======================
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(52) AArch64.NextTableBase(bits(64) descriptor, bit ds, TGx tgx)
bits(52) tablebase = Zeros();
case tgx of
    when TGx_4KB tablebase<47:12> = descriptor<47:12>;
    when TGx_16KB tablebase<47:14> = descriptor<47:14>;
    when TGx_64KB tablebase<47:16> = descriptor<47:16>;
if Have52BitPAExt() && tgx == TGx_64KB then
    tablebase<51:12> = descriptor<15:12>;
elsif ds == '1' then
    tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;
return tablebase;
library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PageBase

// AArch64.PageBase()
// ================
// Extract the address embedded in a page descriptor pointing to the base of
// a memory page

bits(52) AArch64.PageBase(bits(64) descriptor, bit ds, TGx tgx)
bits(52) pagebase = Zeros();

case tgx of
    when TGx_4KB pagebase<47:12> = descriptor<47:12>;
    when TGx_16KB pagebase<47:14> = descriptor<47:14>;
    when TGx_64KB pagebase<47:16> = descriptor<47:16>;

if Have52BitPAExt() && tgx == TGx_64KB then
    pagebase<51:48> = descriptor<15:12>;
elsif ds == '1' then
    pagebase<51:48> = descriptor<9:8>:descriptor<49:48>;
return pagebase;

library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize

// AArch64.PhysicalAddressSize()
// =============================
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bits(3) encoded_ps, TGx tgx)
integer ps;
integer max_ps;

case encoded_ps of
    when '000' ps = 32;
    when '001' ps = 36;
    when '010' ps = 40;
    when '011' ps = 42;
    when '100' ps = 44;
    when '101' ps = 48;
    when '110' ps = 52;
    otherwise
        ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if tgx != TGx_64KB && !Have52BitIPAAndPASpaceExt() then
    max_ps = Min(48, AArch64.PAMax());
else
    max_ps = AArch64.PAMax();
return Min(ps, max_ps);

library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S1StartLevel

// AArch64.S1StartLevel()
// ======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
// Input Address size
// iasize   = AArch64.IASize(walkparams.txsz);
// granulebits = TGxGranuleBits(walkparams.tgx);
// stride   = granulebits - 3;

return FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2SLTTEntryAddress

// AArch64.S2SLTTEntryAddress()
// ============================
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(52) ipa, FullAddress tablebase)

    startlevel = AArch64.S2StartLevel(walkparams);
    isize = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride = granulebits - 3;
    levels = FINAL_LEVEL - startlevel;

    bits(52) index;
    lsb = levels*stride + granulebits;
    msb = isize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(3));

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2StartLevel

// AArch64.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)

    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
            endcase;
        endcase;
        when TGx_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
            endcase;
        endcase;
        when TGx_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
            endcase;
    endcase;
Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTBaseAddress

// AArch64.TTBaseAddress()
// =======================
// Retrieve the PA/IPA pointing to the base of the initial translation table

bits(52) AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz, bits(3) ps, bit ds, TGx tgx, integer startlevel)

    bits(52) tablebase = Zeros();

    // Input Address size
    isize = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride = granulebits - 3;
    levels = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (isize - (levels*stride + granulebits)) + 3;

    if (Have52BitPAExt() && tgx == TGx_64KB && ps == '110') || (ds == '1') then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttb<5:2>:ttb<47:6>;
    else
        tablebase<47:1> = ttb<47:1>;
    end

    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress

// AArch64.TTEntryAddress()
// ========================
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, TGx tgx, bits(6) txsz, bits(64) ia, FullAddress tablebase)

    // Input Address size
    isize = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride = granulebits - 3;
    levels = FINAL_LEVEL - level;

    bits(52) index;
    lsb = levels*stride + granulebits;
    msb = Min(isize - 1, (lsb + stride) - 1);
    index = ZeroExtend(ia<msb:lsb>:Zeros(3));

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
// AArch64.AddrTop()
// ================
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccType acctype, bit tbi)
    if tbid == '1' && acctype == AccType_IFETCH then
        return 63;
    if tbi == '1' then
        return 55;
    else
        return 63;

// AArch64.ContiguousBitFaults()
// =============================
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize  = TranslationSize(tgx, level) + ContiguousSize(tgx, level);
    fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit";
    return tsize > iasize && fault;

// AArch64.DebugFault()
// ====================
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)
    FaultRecord fault;
    fault.statuscode = Fault_Debug;
    fault.acctype    = acctype;
    fault.write      = iswrite;
    fault.secondstage = FALSE;
    fault.s2fs1walk  = FALSE;
    return fault;

// AArch64.ExclusiveFault()
// ========================
FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)
    FaultRecord fault;
    fault.statuscode = Fault_Exclusive;
    fault.acctype    = acctype;
    fault.write      = iswrite;
    fault.secondstage = secondstage;
    fault.s2fs1walk  = s2fs1walk;
    return fault;
// AArch64.IPAIsOutOfRange()
// ================
// Check bits not resolved by translation are ZERO
boolean AArch64.IPAIsOutOfRange(bits(52) ipa, S2TTWParams walkparams)
  // Input Address size
  iasize = AArch64.IASize(walkparams.txsz);
  if iasize < 52 then
    return !IsZero(ipa<51:iasize>);
  else
    return FALSE;

// AArch64.OAOutOfRange()
// =============
// Returns whether output address is expressed in the configured size number of bits
boolean AArch64.OAOutOfRange(TTWState walkstate, bits(3) ps, TGx tgx, bits(64) ia)
  // Output Address size
  oasize = AArch64.PhysicalAddressSize(ps, tgx);
  if oasize < 52 then
    if walkstate.istable then
      baseaddress = walkstate.baseaddress.address;
      return !IsZero(baseaddress<51:oasize>);
    else
      // Output address
      oa = StageOA(ia, tgx, walkstate);
      return !IsZero(oa.address<51:oasize>);
  else
    return FALSE;

// AArch64.S1HasAlignmentFault()
// =============
// Returns whether stage 1 output fails alignment requirement on data accesses to Device memory
boolean AArch64.S1HasAlignmentFault(AccType acctype, boolean aligned, bit ntlsmd, MemoryAttributes memattrs)
  if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
    return FALSE;
  if acctype == AccType_A32LSMD && ntlsmd == '0' && memattrs.device != DeviceType_GRE then
    return TRUE;
  return !aligned || acctype == AccType_DCZVA;
Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1HasPermissionsFault
// AArch64.S1HasPermissionsFault()
// ===============================
// Returns whether stage 1 access violates permissions of target memory

boolean AArch64.S1HasPermissionsFault(Regime regime, SecurityState ss, TTWState walkstate, 
SITTWParams walkparams, boolean ispriv, AccType acctype, 
boolean iswrite) 

bit r;
bit w;
bit x;
permissions = walkstate.permissions;

if HasUnprivileged(regime) then 
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    // Apply leaf permissions
    case permissions.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only
    // Apply hierarchical permissions
    case permissions.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
        when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access
    // Locations writable by unprivileged cannot be executed by privileged
    px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
    ux = NOT(permissions.uxn OR permissions.uxn_table);
    pan access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT, AccType_NV2REGISTER});
    if HavePANEExt() && pan_access && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
        bit pan;
        if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" 
CurrentSecurityState() == SS_Secure 
walkstate.baseaddress.paspace == PAS_NonSecure 
walkparams.sif == '1') then
            ux = '0';
            pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);
        (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only
        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (r,w) = ( r , w ); // No effect
            when '1' (r,w) = ( r ,'0'); // Read-only
        x = NOT(permissions.xn OR permissions.xn_table);
    // Prevent execution from writable locations if WXN is set
    x = x AND NOT(walkparams.wx AND w);
    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if ss == SS_Secure 
walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
    if acctype == AccType_IFETCH then
        if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&
Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1InvalidTxSZ

// AArch64.S1InvalidTxSZ()
// ================
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S1InvalidTxSZ(S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault

// AArch64.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccType acctype, boolean aligned, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;
    return !aligned || acctype == AccType_DCZVA;
// AArch64.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch64.S2HasPermissionsFault(boolean s2fs1walk, TTWState walkstate, SecurityState ss, S2TTWParams walkparams, boolean ispriv, AccType accctype, boolean iswrite)

permissions = walkstate.permissions;
memtype = walkstate.memattrs.memtype;

r = permissions.s2ap<0>;
w = permissions.s2ap<1>;

bit px;
bit ux;
case (permissions.s2xn:permissions.s2xnx) of
  when '00' (px,ux) = ('1','1');
  when '01' (px,ux) = ('0','1');
  when '10' (px,ux) = ('0','0');
  when '11' (px,ux) = ('1','0');

x = if ispriv then px else ux;

if s2fs1walk && walkparams.ptw == '1' && memtype == MemType_Device then
  return TRUE;
elif acctype == AccType_IFETCH then
  constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
  if constraint == Constraint_FAULT & memtype == MemType_Device then
    return TRUE;
  return x == '0';
elif acctype == AccType_DC then
  // AArch32 DC maintenance instructions operating by VA cannot fault.
  if iswrite then
    return !ELUsingAAArch32(EL1) && w == '0';
  else
    return (!ispriv && !ELUsingAAArch32(EL1) && r == '0') ||
    (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elif acctype == AccType_IC then
  // IC instructions do not write
  assert !iswrite;
  impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";
  return (!ispriv && !ELUsingAAArch32(EL1) && r == '0' && impdef_ic_fault) ||
  (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elif iswrite then
  return w == '0';
else
  return r == '0';
// AArch64.S2InconsistentSL()
// =========================
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch64.S2StartLevel(walkparams);
    levels = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (levels * stride // Bits resolved by table walk, except initial level
                     + granulebits // Bits directly mapped to output address
                     + 1); // At least 1 more bit to be decoded by initial level

    // Can accommodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;

    // Configured Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;

// AArch64.S2InvalidSL()
// ---------------------
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '1x1' return TRUE;
                when '11x' return TRUE;
                when '010' return AArch64.PAMax() < 44;
                when '011' return !HaveSmallTranslationTableExt();
                otherwise return FALSE;
        when TGx_16KB
            case walkparams.ds:walkparams.sl0 of
                when '011' return TRUE;
                when '010' return AArch64.PAMax() < 42;
                otherwise return FALSE;
        when TGx_64KB
            case walkparams.sl0 of
                when '11' return TRUE;
                when '10' return AArch64.PAMax() < 44;
                otherwise return FALSE;

// AArch64.S2InvalidTxSZ()
// ------------------------
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S2InvalidTxSZ(S2TTWParams walkparams, boolean s1aarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;
Library pseudocode for aarch64/translation/vmsa_faults/AArch64.VAIsOutOfRange

// AArch64.VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va, AccType accctype, Regime regime, S1TTWParams walkparams)
    addrtop = AArch64.AddrTop(walkparams.tbid, accctype, walkparams.tbi);
    // Input Address size
    iasize = AArch64.IASize(walkparams.txsz);
    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            return !IsZero(va<addrtop:iasize>);
        else
            return !IsOnes(va<addrtop:iasize>);
        else
            return !IsZero(va<addrtop:iasize>);

Library pseudocode for aarch64/translation/vmsa_memattr/AArch64.IsS2ResultTagged

// AArch64.IsS2ResultTagged()
// ==========================
// Determine whether the combined output memory attributes of stage 1 and stage 2 indicate tagged memory

boolean AArch64.IsS2ResultTagged(MemoryAttributes s2_memattrs, boolean s1_tagged)
    return (
        s1_tagged &&
        (s2_memattrs.memtype == MemType_Normal) &&
        (s2_memattrs.inner.attrs == MemAttr_WB) &&
        (s2_memattrs.inner.hints == MemHint_RWA) &&
        (!s2_memattrs.inner.transient) &&
        (s2_memattrs.outer.attrs == MemAttr_WB) &&
        (s2_memattrs.outer.hints == MemHint_RWA) &&
        (!s2_memattrs.outer.transient)
    );
Library pseudocode for aarch64/translation/vmsa_memattr/AArch64.S2ApplyFWBMemAttr

// AArch64.S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs,
                                         bits(4) s2_attr, bits(2) s2_sh)

    MemoryAttributes memattrs;
    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = DecodeDevice(s2_attr<1:0>);
        memattrs.memtype = MemType_Device;
        if s1_memattrs.memtype == MemType_Device then
            memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
        else
            memattrs.device = s2_device;
    elsif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        memattrs = s1_memattrs;
    elsif s2_attr<1:0> == '10' then    // Force writeback
        memattrs.memtype = MemType_Normal;
        memattrs.inner.attrs = MemAttr_WB;
        memattrs.outer.attrs = MemAttr_WB;
        if (s1_memattrs.memtype == MemType_Normal &&
            s1_memattrs.inner.attrs != MemAttr_NC) then
            memattrs.inner.hints = s1_memattrs.inner.hints;
            memattrs.inner.transient = s1_memattrs.inner.transient;
        else
            memattrs.inner.hints = MemHint_RWA;
            memattrs.inner.transient = FALSE;
        
        if (s1_memattrs.outer.attrs != MemAttr_NC) then
            memattrs.outer.hints = s1_memattrs.outer.hints;
            memattrs.outer.transient = s1_memattrs.outer.transient;
        else
            memattrs.outer.hints = MemHint_RWA;
            memattrs.outer.transient = FALSE;
    
    else                               // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType_Device then
            memattrs = s1_memattrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr_NC;
            memattrs.memtype = MemType_Normal;
            memattrs.inner = cacheability_attr;
            memattrs.outer = cacheability_attr;

            s2_shareability = DecodeShareability(s2_sh);
            memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
            memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

            memattrs.shareability = EffectiveShareability(memattrs);
            return memattrs;
### Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.GetS1TLBContext

// AArch64.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)

    TLBContext tlbcontext;

    case regime of
        when Regime_EL3
            tlbcontext = AArch64.TLBContextEL3(ss, va, tg);
        when Regime_EL2
            tlbcontext = AArch64.TLBContextEL2(ss, va, tg);
        when Regime_EL20
            tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
        when Regime_EL10
            tlbcontext = AArch64.TLBContextEL10(ss, va, tg);

    tlbcontext.includes_s1 = TRUE;

    The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;

---

### Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.GetS2TLBContext

// AArch64.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)

    assert EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid = VMID[];
    tlbcontext.tg = tg;
    tlbcontext.ia = ZeroExtend(ipa.address);

    if HaveCommonNotPrivateTransExt() then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    return tlbcontext;
// AArch64.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)

    TLBContext tlbcontext;
    tlbcontext.ss = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid = VMID[];
    tlbcontext.asid = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    tlbcontext.tg = tg;
    tlbcontext.ia = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cn = TTBR0_EL1.CnP;
        else
            tlbcontext.cn = TTBR1_EL1.CnP;
        else
            tlbcontext.cn = '0';
    return tlbcontext;

// AArch64.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)

    TLBContext tlbcontext;
    tlbcontext.ss = ss;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.tg = tg;
    tlbcontext.ia = va;
    tlbcontext.cn = if HaveCommonNotPrivateTransExt() then TTBR0_EL2.CnP else '0';

    return tlbcontext;

// AArch64.TLBContextEL20()
// ========================
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)

    TLBContext tlbcontext;
    tlbcontext.ss = ss;
    tlbcontext.regime = Regime_EL20;
    tlbcontext.asid = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    tlbcontext.tg = tg;
    tlbcontext.ia = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cn = TTBR0_EL2.CnP;
        else
            tlbcontext.cn = TTBR1_EL2.CnP;
        else
            tlbcontext.cn = '0';
    return tlbcontext;
// AArch64.TLBContextEL3()
// =======================
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)

TLBContext tlbcontext;

  tlbcontext.ss = ss;
  tlbcontext.regime = Regime_EL3;
  tlbcontext.tg = tg;
  tlbcontext.ia = va;
  tlbcontext.cnp = if HaveCommonNotPrivateTransExt() then TTBR0_EL3.CnP else '0';

return tlbcontext;

// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)

  if acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM} then
    return EL0;
  elsif acctype == AccType_NV2REGISTER then
    return EL2;
  else
    return PSTATE.EL;

// AArch64.FaultAllowsSetAccessFlag()
// ==================================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.FaultAllowsSetAccessFlag(FaultRecord fault)

  if fault.statuscode == Fault_None then
    return TRUE;
  elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
    return ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE;
  else
    return FALSE;
Library pseudocode for aarch64/translation/vmsa_translation/AArch64.FullTranslate

// AArch64.FullTranslate()
// =======================
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype, boolean iswrite, boolean aligned)

fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM});
regime = TranslationRegime(PSTATE.EL, acctype);
ss = SecurityStateAtEL(PSTATE.EL);

AddressDescriptor ipa;
(fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
  return CreateFaultyAddressDescriptor(va, fault);

if regime == Regime_EL10 && EL2Enabled() then
  s1aarch64 = TRUE;
s2fs1walk = FALSE;
  AddressDescriptor pa;
  (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype, aligned, iswrite, ispriv);

  if fault.statuscode != Fault_None then
    return CreateFaultyAddressDescriptor(va, fault);
  else
    return pa;
else
  return ipa;
Library pseudocode for aarch64/translation/vmsa_translation/AArch64.MemSwapTableDesc

// AArch64.MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(64)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(64) prev_desc, bits(64) new_desc, bit ee, AddressDescriptor descupdateaddress)

descupdateaccess = CreateAccessDescriptor(AccType_ATOMICRW);
FaultRecord fault = fault_in;

// All observers in the shareability domain observe the
// following memory read and write accesses atomically.
(memstatus, mem_desc) = PhysMemRead(descupdateaddress, 8, descupdateaccess);
if ee == '1' then
    mem_desc = BigEndianReverse(mem_desc);
if IsFault(memstatus) then
    iswrite = FALSE;
fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess, 8, fault);
if IsFault(fault.statuscode) then
    fault.acctype = AccType_ATOMICRW;
    return (fault, bits(64) UNKNOWN);

if mem_desc == prev_desc then
    ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
    memstatus = PhysMemWrite(descupdateaddress, 8, descupdateaccess, ordered_new_desc);

    if IsFault(memstatus) then
        iswrite = TRUE;
fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess, 8, fault);
    fault.acctype = memstatus.acctype;
    if IsFault(fault.statuscode) then
        fault.acctype = AccType_ATOMICRW;
        return (fault, bits(64) UNKNOWN);

    // Reflect what is now in memory (in little endian format)
    mem_desc = new_desc;

return (fault, mem_desc);
// AArch64.S1DisabledOutput()
// =========================
// Map the the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime, SecurityState ss, bits(64) va, AccType acctype, boolean aligned)

walkparams = AArch64.GetS1TTWParams(regime, va);
FaultRecord fault = fault_in;

// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

// Output Address
FullAddress oa;
oa.address = va<51:0>;
case ss of
  when SS_Secure     oa.paspace = PAS_Secure;
  when SS_NonSecure  oa.paspace = PAS_NonSecure;
MemoryAttributes memattrs;
if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
  MemAttrHints default_cacheability;
  default_cacheability.attrs          = MemAttr_WB;
  default_cacheability.hints          = MemHint_RWA;
  default_cacheability.transient      = FALSE;
  memattrs.memtype        = MemType_Normal;
  memattrs.outer          = default_cacheability;
  memattrs.inner          = default_cacheability;
  memattrs.shareability   = Shareability_NSH;
  memattrs.tagged         = walkparams.dct == '1';
  memattrs.xs             = '0';
elsif acctype == AccType_IFETCH then
  MemAttrHints i_cache_attr;
  if AArch64.S1ICacheEnabled(regime) then
    i_cache_attr.attrs          = MemAttr_WT;
    i_cache_attr.hints          = MemHint_RA;
    i_cache_attr.transient      = FALSE;
  else
    i_cache_attr.attrs          = MemAttr_NC;
  end if
  memattrs.memtype        = MemType_Normal;
  memattrs.outer          = i_cache_attr;
  memattrs.inner          = i_cache_attr;
  memattrs.shareability   = Shareability_OSH;
  memattrs.tagged         = FALSE;
  memattrs.xs             = '1';
else
  memattrs.memtype        = MemType_Device;
  memattrs.device         = DeviceType_nGnRnE;
  memattrs.shareability   = Shareability_OSH;
  memattrs.xs             = '1';
end if

fault.level = 0;
addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
if !IsZero(va<addrtop:AArch64.PAMax>()) then
  fault.statuscode = Fault_AddressSize;
elsif AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, memattrs) then
  fault.statuscode = Fault_Alignment;
else
  if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);
  end if
  ipa = CreateAddressDescriptor(va, oa, memattrs);
  return (fault, ipa);
// AArch64.S1Translate()
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime, SecurityState ss, bits(64) va, AccType acctype, boolean aligned_in, boolean iswrite_in, boolean ispriv)

FaultRecord fault = fault_in;
boolean aligned = aligned_in;
boolean iswrite = iswrite_in;
// Prepare fault fields in case a fault is detected
fault.secondstage = FALSE;
fault.s2fs1walk   = FALSE;

if !AArch64.S1Enabled(regime) then
  return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
walkparams = AArch64.GetS1TTWParams(regime, va);

if (AArch64.S1InvalidTxSZ(walkparams) ||
    (!ispriv && walkparams.e0pd == '1') ||
    (!ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT) ||
    AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
  fault.statuscode = Fault_Translation;
fault.level      = 0;
return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
  (fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime, ss, acctype, iswrite, ispriv);

if fault.statuscode != Fault_None then
  return (fault, AddressDescriptor UNKNOWN);
if acctype == AccType_IFETCH then
  // Flag the fetched instruction is from a guarded page
  SetInGuardedPage(walkstate.guardedpage == '1');
if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, walkstate.memattrs) then
  fault.statuscode = Fault_Alignment;
elsif IsAtomicRW(acctype) then
  if AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv, acctype, FALSE) then
    // The permission fault was not caused by lack of write permissions
    fault.statuscode = Fault_Permission;
fault.write = FALSE;
elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv, acctype, TRUE) then
    // The permission fault _was_ caused by lack of write permissions
    fault.statuscode = Fault_Permission;
fault.write = TRUE;
elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv, acctype, iswrite) then
  fault.statuscode = Fault_Permission;

  new_desc = descriptor;
  if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
    // Set descriptor AF bit
    new_desc<10> = '1';

  // If HW update of dirty bit is enabled, the walk state permissions
  // will already reflect a configuration permitting writes.
  // The update of the descriptor occurs only if the descriptor bits in
if (fault.statuscode == Fault_None &
  walkparams.ha == '1' &&
  walkparams.hd == '1' &&
  descriptor<51> == '1' && // Descriptor DBM bit
  (IsAtomicRW(acctype) || iswrite) &&
  ![acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC}]) then
  // Clear descriptor AP[2] bit permitting stage 1 writes
  new_desc<7> = '0';

AddressDescriptor descupdateaddress;
FaultRecord s2fault;
// Either the access flag was clear or AP<2> is set
if new_desc != descriptor then
  if regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
    s2fs1walk = TRUE;
    aligned = TRUE;
    iswrite = TRUE;
    (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress, slaarch64,
                                                    ss, s2fs1walk, AccType_ATOMICRW,
                                                    aligned, iswrite, ispriv);

  if s2fault.statuscode != Fault_None then
    return (s2fault, AddressDescriptor UNKNOWN);
  else
    descupdateaddress = descaddress;

  (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                               walkparams.ee, descupdateaddress);

  until new_desc == descriptor || mem_desc == new_desc;

if fault.statuscode != Fault_None then
  return (fault, AddressDescriptor UNKNOWN);

// Output Address
oa = StageOA(va, walkparams.tgx, walkstate);
MemoryAttributes memattrs;
if (acctype == AccType_IFETCH &&
  (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) then
  memattrs = NormalNCMemAttr();
  memattrs.xs = walkstate.memattrs.xs;
elsif (acctype != AccType_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
      walkstate.memattrs.memtype == MemType_Normal) then
  memattrs = NormalNCMemAttr();
  memattrs.xs = walkstate.memattrs.xs;

// The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
// on the Tagged attribute
if HaveMTE2Ext() && walkstate.memattrs.tagged then
  memattrs.tagged = ConstrainUnpredictableBool(Unpredictable_SICTAGGED);
else
  memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    ![boolean IMPLEMENTATION DEFINED "Apply effective shareability at stage 1"]() then
  memattrs.shareability = walkstate.memattrs.shareability;
else
  memattrs.shareability = EffectiveShareability(memattrs);

if acctype == AccType_ATOMICS64 && memattrs.motype == MemType_Normal then
  if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs != MemAttr_NC then
    fault.statuscode = Fault_Exclusive;
    return (fault, AddressDescriptor UNKNOWN);
ipa = CreateAddressDescriptor(va, oa, memattrs);
return (fault, ipa);
// AArch64.S2Translate()
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa, boolean slaarch64, SecurityState ss, boolean s2fs1walk, AccType acctype, boolean aligned, boolean iswrite, boolean ispriv)

walkparams = AArch64.GetS2TTWParams(ss, ipa.paddress.paspace, slaarch64);
FaultRecord fault = fault_in;

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None; // Ignore any faults from stage 1
fault.secondstage = TRUE;
fault.s2fs1walk = s2fs1walk;
fault.ipaddress = ipa.paddress;

if walkparams.vm != '1' then // Stage 2 translation is disabled
  return (fault, ipa);
if (AArch64.S2InvalidTxSZ(walkparams, slaarch64) ||
    AArch64.S2InvalidSL(walkparams) ||
    AArch64.S2InconsistentSL(walkparams) ||
    AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
  fault.statuscode = Fault_Translation;
fault.level = 0;
return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
  (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams, ss, acctype, iswrite, slaarch64);
  if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);
  if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
  elsif IsAtomicRW(acctype) then
    if AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv, acctype, FALSE) then
      // The permission fault was not caused by lack of write permissions
      fault.statuscode = Fault_Permission;
      fault.write = FALSE;
    elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv, acctype, TRUE) then
      // The permission fault _was_ caused by lack of write permissions.
      // However, HW updates, which are atomic writes for stage 1
      // descriptors, permissions fault reflect the original access.
      fault.statuscode = Fault_Permission;
      if !fault.s2fs1walk then
        fault.write = TRUE;
      elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;
      new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
      // Set descriptor AF bit
      new_desc<10> = '1';
    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
// memory do not reflect that and the access instigates a write.
if (fault.statuscode == Fault_None &&
    walkparams.ha == '1' &&
    walkparams.hd == '1' &&
    descriptor<51> == '1' && // Descriptor DBM bit
    (IsAtomicRW(acctype) || iswrite) &&
    !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
    // Set descriptor S2AP[1] bit permitting stage 2 writes
    new_desc<7> = '1';

    // Either the access flag was clear or S2AP<1> is clear
    if new_desc != descriptor then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
            walkparams.ee, descaddress);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.tgx, walkstate);
    MemoryAttributes s2_memattrs;
    if ((s2fs1walk &&
             walkstate.memattrs.memtype == MemType_Device &&
             walkparams.ptw == '0') ||
        (acctype == AccType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Device ||
             HCR_EL2.ID == '1')) ||
        (acctype != AccType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal &&
             HCR_EL2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;

    if !s2fs1walk && acctype == AccType_ATOMICLS64 && s2_memattrs.memtype == MemType_Normal then
        if s2_memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.attrs != MemAttr_NC then
            fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    MemoryAttributes memattrs;
    if walkparams.fwb == '0' then
        memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
    else
        memattrs = s2_memattrs;

    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
    return (fault, pa);

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.TranslateAddress

// AArch64.TranslateAddress()
// -------------------------------------------
// Main entry point for translating an address
AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype, boolean iswrite, boolean aligned, integer size)
result = AArch64.FullTranslate(va, acctype, iswrite, aligned);
if !IsFault(result) then
    result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va);
return result;
### Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlockDescSupported

```java
// AArch64.BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit ds, TGx tgx, integer level)
{
    case tgx of
        when TGx_4KB return level == 2 || level == 1 || (level == 0 && ds == '1');
        when TGx_16KB return level == 2 || (level == 1 && ds == '1');
        when TGx_64KB return level == 2 || (level == 1 && AArch64.PAMax() == 52);
    return FALSE;
}
```

### Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlocknTFaults

```java
// AArch64.BlocknTFaults()
// =======================
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bits(64) descriptor)
{
    if !HaveBlockBBM() then
        return FALSE;
    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = boolean IMPLEMENTATION_DEFINED 'BBM level 1 or 2 support nT bit causes Translation Fault';
    return bbm_level IN {1, 2} && descriptor<16> == '1' && nT_faults;
}
```

### Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.ContiguousBit

```java
// AArch64.ContiguousBit()
// =======================
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, integer level, bits(64) descriptor)
{
    if tgx == TGx_64KB && level == 1 && !Have52BitVAExt() then
        return '0'; // RES0
    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0
    if tgx == TGx_4KB && level == 0 then
        return '0'; // RES0
    return descriptor<52>;
}
```

### Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.DecodeDescriptorType

```java
// AArch64.DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(64) descriptor, bit ds, TGx tgx, integer level)
{
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' then
        if AArch64.BlockDescSupported(ds, tgx, level) then
            return DescriptorType_Block;
        else
            return DescriptorType_Invalid;
        end
    else
        return DescriptorType_Invalid;
    end
}
```
Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyOutputPerms

```c
// AArch64.S1ApplyOutputPerms()
// -----------------------------------------------
// Apply output permissions encoded in stage 1 page/block descriptors
Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(64) descriptor, Regime regime, S1TTWParams walkparams) {
    Permissions permissions = permissions_in;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        permissions.ap<2:1> = descriptor<7>:'0';
        permissions.pxn = descriptor<54>;
    elsif HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn = descriptor<54>;
        permissions.pxn = descriptor<53>;
    else
        permissions.ap<2:1> = descriptor<7>:'1';
        permissions.xn = descriptor<54>;
    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no permission faults caused by lack of write permissions are reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.ap<2> = '0';
    return permissions;
}
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyTablePerms

```c
// AArch64.S1ApplyTablePerms()
// ------------------------------
// Apply hierarchical permissions encoded in stage 1 table descriptors
Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(64) descriptor, Regime regime, S1TTWParams walkparams) {
    Permissions permissions = permissions_in;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table  = descriptor<62>:'0';
        pxn_table = descriptor<60>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    elsif HasUnprivileged(regime) then
        ap_table  = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor<62>:'0';
        xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;
    return permissions;
}
```
Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S2ApplyOutputPerms

// AArch64.S2ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(64) descriptor, S2TTWParams walkparams)

  Permissions permissions;
  permissions.s2ap = descriptor<7:6>;
  permissions.s2xn = descriptor<54>;

  if HaveExtendedExecuteNeverExt() then
    permissions.s2xnx = descriptor<53>;
  else
    permissions.s2xnx = '0';

  // Descriptors marked with DBM set have the effective value of S2AP[1] set.
  // This implies no permission faults caused by lack of write permissions are
  // reported, and the Dirty bit can be set.
  if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
    permissions.s2ap<1> = '1';

  return permissions;

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1InitialTTWState

// AArch64.S1InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss)

  TTWState walkstate;
  FullAddress tablebase;
  Permissions permissions;

  startlevel  = AArch64.S1StartLevel(walkparams);
  ttbr        = AArch64.S1TTBR(regime, va);
  case ss of
    when SS_Secure  tablebase.paspace = PAS_Secure;
    when SS_NonSecure tablebase.paspace = PAS_NonSecure;

  tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds, walkparams.tgx, startlevel);

  permissions.ap_table = Zeros();
  if HasUnprivileged(regime) then
    permissions.uxn_table = Zeros();
    permissions.pxn_table = Zeros();
  else
    permissions.xn_table  = Zeros();

  walkstate.baseaddress = tablebase;
  walkstate.level       = startlevel;
  walkstate.istable     = TRUE;
  // In regimes that support global and non-global translations, translation
  // table entries from lookup levels other than the final level of lookup
  // are treated as being non-global
  walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
  walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
  walkstate.permissions = permissions;

  return walkstate;
Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLast

// AArch64.S1NextWalkStateLast()
// =============================
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLast(TTWState currentstate, Regime regime, SecurityState ss, S1TTWParams walkparams, bits(64) descriptor)

    TTWState  nextstate;
    FullAddress baseaddress;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx, currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        baseaddress.paspace = if descriptor<5> == '0' then PAS_Secure else PAS_NonSecure;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    attrindx = descriptor<4:2>;
    sh     = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    attr = MAIRAttr(UInt(attrindx), walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs    = S1DecodeMemAttrs(attr, sh, slaarch64);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions, descriptor, regime, walkparams);
    nextstate.contiguous  = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    if HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure then
        // In Secure state, a translation must be treated as non-global, regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
        nextstate.nG = '1';
    else
        nextstate.nG = descriptor<11>;

    nextstate.guardedpage = descriptor<50>;

    return nextstate;
// AArch64.S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, Regime regime, S1TTWParams walkparams, bits(64) descriptor)

    TTWState nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        tablebase.paspace = if descriptor<63> == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable = TRUE;
    nextstate.nG = currentstate.nG;
    nextstate.level = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs = currentstate.memattrs;

    if walkparams.hpd == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions, descriptor, regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;

    return nextstate;
AArch64.S1Walk()
==============
Traverse stage 1 translation tables obtaining the final descriptor
as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(FaultRecord fault_in, S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss, AccType acctype, boolean iswrite_in, boolean ispriv)

FaultRecord fault = fault_in;
boolean iswrite = iswrite_in;
if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault_Translation;
fault.level = 0;
return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, ss);

// Detect Address Size Fault by TTB
if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
    fault.statuscode = Fault_AddressSize;
fault.level = 0;
return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

bits(64) descriptor;
AddressDescriptor walkaddress;
walkaddress.vaddress = va;
if !AArch64.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx, walkparams.txsz, va, walkstate.baseaddress);
    walkaddress.paddress = descaddress;
    if regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        s2fs1walk = TRUE;
        aligned = TRUE;
        iswrite = FALSE;
    end
    (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, s1aarch64, ss, s2fs1walk, AccType_TTW, aligned, iswrite, ispriv);
    if s2fault.statuscode != Fault_None then
        return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
    end
    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
else
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);
if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
end
desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
case desctype of
    when DescriptorType_Table
        walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams, descriptor);
        // Detect Address Size Fault by table descriptor
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
            fault.statuscode = Fault_AddressSize;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
    end when;
    when DescriptorType_Page, DescriptorType_Block
        walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss, walkparams, descriptor);
    end when;
    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
    otherwise
        Unreachable();
    end case;
until desctype IN {DescriptorType_Page, DescriptorType_Block};
if (walkstate.contiguous == '1' &&
     AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
    fault.statuscode = Fault_Translation;
elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
             !(acctype IN {AccType_DC, AccType_IC} &&
               !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations") then
        fault.statuscode = Fault_AccessFlag;
    return (fault, walkaddress, walkstate, descriptor);
end if;

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2InitialTTWState

AArch64.S2InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 2
TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkparams)
    TTWState walkstate;
    FullAddress tablebase;
    ttbr = VTTBR_EL2;
    startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.paspace = PAS_NonSecure;
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds, walkparams.tgx, startlevel);
    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    return walkstate;
// AArch64.S2NextWalkStateLast()
// =============================
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLast(TTWState currentstate, SecurityState ss, S2TTWParams walkparams, AddressDescriptor ipa, bits(64) descriptor)

    TTWState nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    else
        baseaddress.paspace = PAS_NonSecure;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx, currentstate.level);

    nextstate.istable = FALSE;
    nextstate.level   = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;

    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
        if s2_attr<1:0> == '10' then    // Force writeback
            nextstate.memattrs.xs = '0';
        else
            nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    else
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh);
        nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;

    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    return nextstate;

// AArch64.S2NextWalkStateTable()
// ==============================
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams, bits(64) descriptor)

    TTWState nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable = TRUE;
    nextstate.level   = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs = currentstate.memattrs;

    return nextstate;
AArch64.S2Walk()

Traverse stage 2 translation tables obtaining the final descriptor
as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
    FaultRecord fault in, AddressDescriptor ipa, S2TTWParams walkparams, SecurityState ss,
    AccType acctype, boolean iswrite, boolean slaarch64)

    FaultRecord fault = fault in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate;
    if ss == SS_Secure then
        walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64.S2InitialTTWState(ss, walkparams);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;
    walkaddress.vaddress = ipa.vaddress;
    if HCR_EL2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;
        FullAddress descaddress;
        if walkstate.level == AArch64.S2StartLevel(walkparams) then
            // Initial lookup might index into concatenated tables
            descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address,
                                                        walkstate.baseaddress);
        else
            ipa_64 = ZeroExtend(ipa.paddress.address, 64);
            descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx, walkparams.txsz,
                                                    ipa_64, walkstate.baseaddress);

        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);
        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx, walkstate.level);
        case desctype of
            when DescriptorType_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
                // Detect Address Size Fault by table descriptor
                if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
            when DescriptorType_Page, DescriptorType_Block
                walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,
when DescriptorType_Invalid
    fault.statuscode = Fault_Translation;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

otherwise
    Unreachable();

until desctype IN {DescriptorType_Page, DescriptorType_Block};

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
    fault.statuscode = Fault_Translation;
elsif desctype == DescriptorType_Block &&
    AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
    !(acctype IN {AccType_DC, AccType_IC} &&
    !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations") ) then
    fault.statuscode = Fault_AccessFlag;

return (fault, walkaddress, walkstate, descriptor);

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2InitialTTWState

// AArch64.SS2InitialTTWState()
// ============================
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)
    TTWState walkstate;
    FullAddress tablebase;

    bits(64) ttbr;
    if ipaspace == PAS_Secure then
        ttbr = VSTTBR_EL2;
    else
        ttbr = VTTBR_EL2;

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
        else
            if walkparams.nsw == '0' then
                tablebase.paspace = PAS_Secure;
            else
                tablebase.paspace = PAS_NonSecure;
            else
                tablebase.paspace = PAS_NonSecure;
        startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                             walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    walkstate.istable = TRUE;
    walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
// AArch64.SS2OutputPASpace
// ==========================
// Assign PA Space to output of Secure stage 2 translation

AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
if ipaspace == PAS_Secure
  if walkparams.<sw,sa> == '00' then
    return PAS_Secure;
  else
    return PAS_NonSecure;
else
  if walkparams.<sw,sa,nsw,nsa> == '0000' then
    return PAS_Secure;
  else
    return PAS_NonSecure;

// AArch64.BBMSupportLevel()
// =========================
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
if !HaveBlockBBM() then
  return integer UNKNOWN;
else
  return integer IMPLEMENTATION_DEFINED "Block BBM support level";

// AArch64.DecodeTG0()
// ===================
// Decode granule size configuration bits TG0

AArch64.DecodeTG0(bits(2) tg0_in)
bits(2) tg0 = tg0_in;
if tg0 == '11' then
  tg0 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG0 encoding granule size";
case tg0 of
  when '00' return TGx_4KB;
  when '01' return TGx_64KB;
  when '10' return TGx_16KB;

// AArch64.DecodeTG1()
// ===================
// Decode granule size configuration bits TG1

AArch64.DecodeTG1(bits(2) tg1_in)
bits(2) tg1 = tg1_in;
if tg1 == '00' then
  tg1 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG1 encoding granule size";
case tg1 of
  when '10' return TGx_4KB;
  when '11' return TGx_64KB;
  when '01' return TGx_16KB;
Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS1TTWParams

// AArch64.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// system registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, bits(64) va)
S1TTWParams walkparams;

varange = AArch64.GetVARange(va);

case regime of
  when Regime_EL3 walkparams = AArch64.S1TTWParamsEL3();
  when Regime_EL2 walkparams = AArch64.S1TTWParamsEL2();
  when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(varange);
  when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);

if UInt(walkparams.txsz) > maxtxsz then
  if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
    walkparams.txsz = maxtxsz<5:0>;
  elsif !Have52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
      walkparams.txsz = mintxsz<5:0>;

return walkparams;

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS2TTWParams

// AArch64.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASpace ipaspace, boolean s1aarch64)
S2TTWParams walkparams;

if ss == SS_NonSecure then
  walkparams = AArch64.NSS2TTWParams(s1aarch64);
elsif HaveSecureEL2Ext() && ss == SS_Secure then
  walkparams = AArch64.SS2TTWParams(ipaspace, s1aarch64);
else
  Unreachable();

maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);

if UInt(walkparams.txsz) > maxtxsz then
  if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
    walkparams.txsz = maxtxsz<5:0>;
  elsif !Have52BitPAExt() && UInt(walkparams.txsz) < mintxsz then
    if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
      walkparams.txsz = mintxsz<5:0>;

return walkparams;

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetVARange

// AArch64.GetVARange()
// ====================
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
if va<55> == '0' then
  return VARange_LOWER;
else
  return VARange_UPPER;
// AArch64.MaxTxSZ()
// ===============
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
  if HaveSmallTranslationTableExt() && !UsingAArch32() then
    case tgx of
      when TGx_4KB return 48;
      when TGx_16KB return 48;
      when TGx_64KB return 47;
    return 39;
  end;

// AArch64.NSS2TTWParams()
// ==============
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean slaarch64)
  S2TTWParams walkparams;
  walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
  walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
  walkparams.txsz = VTCR_EL2.T0SZ;
  walkparams.sl0  = VTCR_EL2.SL0;
  walkparams.ps   = VTCR_EL2.PS;
  walkparams.irgn = VTCR_EL2.IRGN0;
  walkparams.orgn = VTCR_EL2.ORGN0;
  walkparams.sh   = VTCR_EL2.SH0;
  walkparams.ee   = SCTLR_EL2.EE;
  walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
  walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
  walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
  walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
  if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = VTCR_EL2.DS;
  else
    walkparams.ds = '0';
  if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
    walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
  else
    walkparams.sl2 = '0';
  walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
  return walkparams;

// AArch64.PAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this processor

integer AArch64.PAMax()
  return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
// AArch64.S1DCacheEnabled()
// =======================
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLR_EL3.C == '1';
        when Regime_EL2 return SCTLR_EL2.C == '1';
        when Regime_EL20 return SCTLR_EL2.C == '1';
        when Regime_EL10 return SCTLR_EL1.C == '1';

// AArch64.S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);
    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;

// AArch64.S1Enabled()
// =================
// Determine if stage 1 for the acting translation regime is enabled

boolean AArch64.S1Enabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLR_EL3.M == '1';
        when Regime_EL2 return SCTLR_EL2.M == '1';
        when Regime_EL20 return SCTLR_EL2.M == '1';
        when Regime_EL10 return (! EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';

// AArch64.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLR_EL3.I == '1';
        when Regime_EL2 return SCTLR_EL2.I == '1';
        when Regime_EL20 return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';

// AArch64.S1MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(bit ds, TGx tgx)
    if (Have52BitVAExt() & tgx == TGx_64KB) || ds == '1' then
        return 12;
    return 16;
// AArch64.S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

bits(64) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
    when Regime_EL3 return TTBR0_EL3;
    when Regime_EL2 return TTBR0_EL2;
    when Regime_EL20 return if varange == VARange_LOWER then TTBR0_EL2 else TTBR1_EL2;
    when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1;
S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)

if varange == VARange_LOWER then
    walkparams.tgx = AArch64.DecodeTG0(TCR_EL1.TG0);
    walkparams.txsz = TCR_EL1.T0SZ;
    walkparams.ign = TCR_EL1.IRGN0;
    walkparams.orgn = TCR_EL1.ORGN0;
    walkparams.sh = TCR_EL1.SH0;
    walkparams.tbi = TCR_EL1.TB10;

    walkparams.nfd = if HaveSVE() then TCR_EL1.NFD0 else '0';
    walkparams.tbid = if HavePACExt() then TCR_EL1.TBID0 else '0';
    walkparams.e0pd = if HaveE0PDExt() then TCR_EL1.E0PD0 else '0';
    walkparams.hpde = if AArch64.HaveHPDExt() then TCR_EL1.HPD0 else '0';
else
    walkparams.tgx = AArch64.DecodeTG1(TCR_EL1.TG1);
    walkparams.txsz = TCR_EL1.T1SZ;
    walkparams.ign = TCR_EL1.IRGN1;
    walkparams.orgn = TCR_EL1.ORGN1;
    walkparams.sh = TCR_EL1.SH1;
    walkparams.tbi = TCR_EL1.TB11;

    walkparams.nfd = if HaveSVE() then TCR_EL1.NFD1 else '0';
    walkparams.tbid = if HavePACExt() then TCR_EL1.TBID1 else '0';
    walkparams.e0pd = if HaveE0PDExt() then TCR_EL1.E0PD1 else '0';
    walkparams.hpde = if AArch64.HaveHPDExt() then TCR_EL1.HPD1 else '0';

walkparams.mair = MAIR_EL1;
walkparams.wxn = SCTLR_EL1.WXN;
walkparams.ps = TCR_EL1.IPS;
walkparams.ee = SCTLR_EL1.EE;
walkparams.sif = SCR_EL3.SIF;

if EL2Enabled() then
    walkparams.dc = HCR_EL2.DC;
    walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';
if HaveTrapLoadStoreMultipleDeviceExt() then
    walkparams.ntlsmd = SCTLR_EL1.nTLSMD;
else
    walkparams.ntlsmd = '1';

if EL2Enabled() then
    if HCR_EL2.<NV,NV1> == '01' then
        case ConstraintUnpredictable(Unpredictable_NVNV1) of
            when Constraint_NVNV1_00 walkparams_nv1 = '0';
            when Constraint_NVNV1_01 walkparams_nv1 = '1';
            when Constraint_NVNV1_11 walkparams_nv1 = '1';
else
    walkparams_nv1 = HCR_EL2.NV1;
else
    walkparams_nv1 = '0';

walkparams.epan = if HavePAN3Ext() then SCTLR_EL1.EPAN else '0';
walkparams.cmow = if HaveFeatCMOW() then SCTLR_EL1.CMOW else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
    walkparams.ds = TCR_EL1.DS;
else
    walkparams.ds = '0';

return walkparams;
AArch64.S1TTWParamsEL2()
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2()

    walkparams.tgx = AArch64.DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGN0;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt() then TCR_EL2.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20

// AArch64.S1TTWParamsEL20()
// =========================
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20( VARange varrange )
    S1TTWParams walkparams;
    if varrange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0( TCR_EL2.TG0 );
        walkparams.txsz = TCR_EL2.T0SZ;
        walkparams.irgn = TCR_EL2.IRGN0;
        walkparams.orgn = TCR_EL2.ORGN0;
        walkparams.sh = TCR_EL2.SH0;
        walkparams.tbi = TCR_EL2.TBI0;
        walkparams.nfd = if HaveSVE() then TCR_EL2.NFD0 else '0';
        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDExt() then TCR_EL2.E0PD0 else '0';
        walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1( TCR_EL2.TG1 );
        walkparams.txsz = TCR_EL2.T1SZ;
        walkparams.irgn = TCR_EL2.IRGN1;
        walkparams.orgn = TCR_EL2.ORGN1;
        walkparams.sh = TCR_EL2.SH1;
        walkparams.tbi = TCR_EL2.TBI1;
        walkparams.nfd = if HaveSVE() then TCR_EL2.NFD1 else '0';
        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDExt() then TCR_EL2.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD1 else '0';
    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ps = TCR_EL2.IPS;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
    else
        walkparams.ntlsmd = '1';
    walkparams.epan = if HavePAN3Ext() then SCTLR_EL2.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW() then SCTLR_EL2.CMOW else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN { TGx_4KB, TGx_16KB } && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    return walkparams;

Shared Pseudocode Functions
// AArch64.S1TTWParamsEL3()
// ========================
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()

S1TTWParams walkparams;
walkparams.tgx = AArch64.DecodeTG0(TCR_EL3.TG0);
walkparams.txsz = TCR_EL3.T0SZ;
walkparams.ps = TCR_EL3.PS;
walkparams.irgn = TCR_EL3.IRGN0;
walkparams.orgn = TCR_EL3.ORGN0;
walkparams.sh = TCR_EL3.SH0;
walkparams.tbi = TCR_EL3.TBI;
walkparams.mair = MAIR_EL3;
walkparams.wxn = SCTLR_EL3.WXN;
walkparams.ee = SCTLR_EL3.EE;
walkparams.sif = SCR_EL3.SIF;
walkparams.tbid = if HavePACExt() then TCR_EL3.TBID else '0';
walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL3.HPD else '0';
walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL3.HA else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL3.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
  walkparams.ds = TCR_EL3.DS;
else
  walkparams.ds = '0';
return walkparams;

// AArch64.S2MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

integer AArch64.S2MinTxSZ(bit ds, TGx tgx, boolean s1aarch64)

ips = AArch64.PAMax();
if Have52BitPAExt() && tgx != TGx_64KB && ds == '0' then
  ips = Min(48, AArch64.PAMax());

min_txsz = 64 - ips;
if !s1aarch64 then
  // EL1 is AArch32
  min_txsz = Min(min_txsz, 24);
return min_txsz;
// AAarch64.SS2TTWParams()
// ======================
// Gather walk parameters specific for secure stage 2 translation

AArch64.SS2TTWParams ipaspace, boolean s1aarch64)
S2TTWParams walkparams;
if ipaspace == PAS_Secure then
  walkparams.tgx = AArch64.DecodeTG0(VSTCR_EL2.TG0);
  walkparams.txsz = VSTCR_EL2.T0SZ;
  walkparams.sl0 = VSTCR_EL2.SL0;
  if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
    walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
  else
    walkparams.sl2 = '0';
else if ipaspace == PAS_NonSecure then
  walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
  walkparams.txsz = VTCR_EL2.T0SZ;
  walkparams.sl0 = VTCR_EL2.SL0;
  if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
    walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
  else
    walkparams.sl2 = '0';
else
  Unreachable();
walkparams.sw = VSTCR_EL2.SW;
walkparams.nsw = VTCR_EL2.NSW;
walkparams.sa = VSTCR_EL2.SA;
walkparams.nsa = VTCR_EL2.NSA;
walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
walkparams.ps = VTCR_EL2.PS;
walkparams.irgn = VTCR_EL2.IRGN0;
walkparams.orgn = VTCR_EL2.ORGN0;
walkparams.sh = VTCR_EL2.SH0;
walkparams.ee = SCTLR_EL2.EE;
walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW else '0';
walkparams.fwb = if HaveStage2MemAttrControl() then VCR_EL2.FW else '0';
walkparams.hd = if HaveDirtyBitModifierExt() then VCR_EL2.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
  walkparams.ds = VTCR_EL2.DS;
else
  walkparams.ds = '0';
walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2.IsEnabled() then HCRX_EL2.CMOW else '0';
return walkparams;

// AAarch64.VAMax()
// ====================
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// the virtual address for this processor
integer AAarch64.VAMax()
  return integer IMPLEMENTATION DEFINED "Maximum Virtual Address Size";
Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================
ClearStickyErrors()
    EDSR.TXU = '0';            // Clear TX underrun flag
    EDSR.RXO = '0';            // Clear RX overrun flag
    if Halted() then            // in Debug state
        EDSR.ITO = '0';        // Clear ITR overrun flag
    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSR.IE == '0' && ConstrUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSR.ERR = '0';            // Clear cumulative error flag
    return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// ================

bits(2) DebugTargetFrom(boolean secure)
    boolean route_to_el2;
    if HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() &&
        ('!HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;
    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elseif HaveEL(EL3) && !HaveAArch64() && secure then
        target = EL3;
    else
        target = EL1;
    return target;
```
Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```java
// DoubleLockStatus()
// ===================
// Returns the state of the OS Double Lock.
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
if !HaveDoubleLock() then
    return FALSE;
elseif ELUsingAArch32(EL1) then
    return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```java
// OSLockStatus()
// ===============
// Returns the state of the OS Lock.

boolean OSLockStatus()
return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

Library pseudocode for shared/debug/SoftwareLockStatus/Component

```java
enumeration Component {
    Component_PMU,
    Component_Debug,
    Component_CTI
}
```

Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```java
// Returns the accessed component.
Component GetAccessComponent();
```

Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```java
// SoftwareLockStatus()
// ======================
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
Component component = GetAccessComponent();
if !HaveSoftwareLock(component) then
    return FALSE;
case component of
    when Component_Debug
        return EDLSR.SLK == '1';
    when Component_PMU
        return PMLSR.SLK == '1';
    when Component_CTI
        return CTILSR.SLK == '1';
    otherwise
        Unreachable();
```

Library pseudocode for shared/debug/authentication/AccessState

```java
// Returns the Security state of the access.
SecurityState AccessState();
```
// AllowExternalDebugAccess()
// =========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalDebugAccess(AccessState());

// AllowExternalDebugAccess()
// =========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;
    if HaveEL(EL3) then
        EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

// AllowExternalPMUAccess()
// =========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess(AccessState());

// AllowExternalPMUAccess()
// =========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;
    if HaveEL(EL3) then
        EPMAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation();

signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

// ExternalInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
return DBGEN == HIGH;

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
if !ExternalInvasiveDebugEnabled() then return FALSE;
ss = SecurityStateAtEL(PSTATE.EL);
if (ELUsingAArch32(EL1) && PSTATE.EL == EL0 &&
  ss == SS_Secure && SDER.SUNIDEN == '1') then
  return TRUE;
case ss of
  when SS_NonSecure return TRUE;
  when SS_Secure return ExternalSecureNoninvasiveDebugEnabled();

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
if HaveNoninvasiveDebugAuth() then
  return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
else
  return ExternalSecureInvasiveDebugEnabled();
Library pseudocode for shared/debug/authentication/IsAccessSecure

// Returns TRUE when an access is Secure
boolean IsAccessSecure();

Library pseudocode for shared/debug/authentication/IsCorePowered

// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behaviour, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC_in, bit HMC_in, bits(2) PxC_in, boolean isbreakpnt)

    boolean reserved = FALSE;
    bits(2) SSC = SSC_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;

    // Values that are not allocated in any architecture version
    if (HMC:SSC:PxC) IN {'01110','100x0','10110','11x10'} then
        reserved = TRUE;

    // Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && HMC:SSC:PxC == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !HaveSecureEL2Ext() && (HMC:SSC:PxC) IN {'01100','10100','x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value
        return (Constraint_NONE, SSC, HMC, PxC);
Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

// NumBreakpointsImplemented()
// retourns the number of breakpoints implemented. This is indicated to software by
// DBGDIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.
integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";

Library pseudocode for shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

// NumContextAwareBreakpointsImplemented()
// returns the number of context-aware breakpoints implemented. This is indicated to software by
// DBGDIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.
integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";

Library pseudocode for shared/debug/breakpoint/NumWatchpointsImplemented

// NumWatchpointsImplemented()
// returns the number of watchpoints implemented. This is indicated to software by
// DBGDIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.
integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);

Library pseudocode for shared/debug/cti/CTI_SignalEvent

// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);

Library pseudocode for shared/debug/cti/CrossTrigger

enumeration CrossTriggerOut  {
    CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
    CrossTriggerOut_IRQ, CrossTriggerOut_RSVDD3,
    CrossTriggerOut_TraceExtIn0, CrossTriggerOut_TraceExtIn1,
    CrossTriggerOut_TraceExtIn2, CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {
    CrossTriggerIn_CrossHalt, CrossTriggerIn_PMUOverflow,
    CrossTriggerIn_RSVDD2, CrossTriggerIn_RSVDD3,
    CrossTriggerIn_TraceExtOut0, CrossTriggerIn_TraceExtOut1,
    CrossTriggerIn_TraceExtOut2, CrossTriggerIn_TraceExtOut3};
Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                    (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                    (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write

if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
    EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
    return;

EDSCR.RXfull = '1';
DTRRX = value;

if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
    if !UsingAArch32() then
        ExecuteA64(0xD5330501<31:0>);                // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>);                // A64 "STR W1,[X0],#4"
        R[1] = bits(64) UNKNOWN;
    else
        ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
        ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
        R[1] = bits(32) UNKNOWN;
    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(64) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_EL0" calls DBGDTRX_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
        return;

// DBGDTRRX_EL0[] (external read)
// ===============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
// DBGDTRTX_EL0[] (external read)
// ----------------------------------------
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

if EDPRSR<6:5,0> !='001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;
end;

underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
value = if underrun then bits(32) UNKNOWN else DTRTX;

if EDSCR.ERR == '1' then return value; // Error flag set: no side-effects

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return value; // Software lock locked: no side-effects

if underrun then
    EDSCR.TXU = '1'; EDSCR.ERR = '1'; // Underrun condition: block side-effects
    return value; // Return UNKNOWN
end;

EDSCR.TXfull = '0';
if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
end;

if !UsingAArch32() then
    ExecuteA64(0xB82044401<31:0>); // A64 "LDR W1,[X0],#4"
else
    ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "LDR R1,[R0],#4"
end;

if EDSCR.ERR == '1' then
    EDSCR.TXfull = bit UNKNOWN;
    DBGDTRTX_EL0 = bits(64) UNKNOWN;
end;

else if !UsingAArch32() then
    ExecuteA64(0x05130501<31:0>); // A64 "MSR DBGDTRTX_EL0,X1"
else
    ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "MSR DBGDTRTXint,R1"
end;

assert EDSCR.TXfull == '1';
if !UsingAArch32() then
    X[1] = bits(64) UNKNOWN;
else
    R[1] = bits(32) UNKNOWN;
    EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
end;

return value;

// DBGDTRTX_EL0[] (external write)
// ---------------------------------------

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;
Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value_in
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ====================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
        EDSCR.RXfull = '0';
        return result;

Library pseudocode for shared/debug/dccanditr/DTR

bits(32) DTRRX;
bits(32) DTRTX;
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
   IMPLEMENTATION_DEFINED "generate error response";
   return;
if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write
if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
if !Halted() then return;                           // Non-debug state: ignore write
if EDSCR.ITE == '0' || EDSCR.MA == '1' then
   EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
   return;
   // ITE indicates whether the processor is ready to accept another instruction; the processor
   // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
   // is no indication that the pipeline is empty (all instructions have completed). In this
   // pseudocode, the assumption is that only one instruction can be executed at a time,
   // meaning ITE acts like "InstrCompl".
   EDSCR.ITE = '0';
if !UsingAArch32() then
   ExecuteA64(value);
else
   ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);
EDSCR.ITE = '1';
return;
Library pseudocode for shared/debug/halting/DCPSInstruction
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EU == EL2 || (PSTATE.EU == EL3 && UsingAArch32()) then handle_el = PSTATE.EU;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
            else handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then UNDEFINED;
            elsif PSTATE.EU == EL3 && !UsingAArch32() then handle_el = EL3;
            elseif IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
            handle_el = EL3;
        otherwise
            Unreachable();
    end;

    from_secure = IsSecure();
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32(); // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
                when EL2
                    AArch32.WriteMode(M32_Hyp);
                when EL3
                    AArch32.WriteMode(M32_Monitor);
                if HavePANExt() then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTLR.SPAN == '0' then
                        PSTATE.PAN = '1';
                    if handle_el == EL2 then
                        ELR.hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
                    else
                        LR = bits(32) UNKNOWN;
                        SPSR[] = bits(32) UNKNOWN;
                        PSTATE.E = SCTLR[].EE;
                        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
                    end;
            /// Targeting AArch64
            end;
        end;
        if UsingAArch32() then
            AArch64.MaybeZeroRegisterUppers();
            MaybeZeroSVEUppers(target_el);
            PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EU = handle_el;
            if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
                PSTATE.PAN = '1';
            ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(64) UNKNOWN;  ESR[] = bits(64) UNKNOWN;
            DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
            if HaveIAOEExt() then PSTATE.UAO = '0';
            if HaveMTEExt() then PSTATE.TCO = '1';
        UpdateEDSCRFields(); // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EU == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
if sync_errors then
    SynchronizeErrors();
return;

Library pseudocode for shared/debug/halting/DRPSInstruction

// DRPSInstruction()
// ==============
// Operation of the A64 DRPS and T32 ERET instructions in Debug state
DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
if HaveDoubleFaultExt() && !UsingAArch32() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
// SCTLR[].IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
if sync_errors then
    SynchronizeErrors();
DebugRestorePSR();
return;

Library pseudocode for shared/debug/halting/DebugHalt

const bits(6) DebugHalt_Breakpoint = '000111';
const bits(6) DebugHalt_EDBGRQ = '010011';
const bits(6) DebugHalt_Step_Normal = '011011';
const bits(6) DebugHalt_Step_Exclusive = '011111';
const bits(6) DebugHalt_OSUnlockCatch = '100011';
const bits(6) DebugHalt_ResetCatch = '100111';
const bits(6) DebugHalt_Watchpoint = '101011';
const bits(6) DebugHalt_HaltInstruction = '101111';
const bits(6) DebugHalt_SoftwareAccess = '110011';
const bits(6) DebugHalt_ExceptionCatch = '110111';
const bits(6) DebugHalt_Step_NoSyndrome = '111011';

Library pseudocode for shared/debug/halting/DebugRestorePSR

// DebugRestorePSR()
// ===============
DebugRestorePSR()

    // PSTATE.{N,Z,C,V,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        bits(32) spsr = SPSR[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,GE,SS,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';  // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        bits(64) spsr = SPSR[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
        UpdateEDSCRFields();  // Update ESCR PE state flags
Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

DisableITRAndResumeInstructionPrefetch();

Library pseudocode for shared/debug/halting/ExecuteA64

// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);

Library pseudocode for shared/debug/halting/ExecuteT32

// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);

Library pseudocode for shared/debug/halting/ExitDebugState

// ExitDebugState()
// ================
ExitDebugState()
assert Halted();
SynchronizeContext();

// Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
detected that the PE has restarted.
EDSCR.STATUS = '000001';                           // Signal restarting
EDESR<2:0> = '000';                                // Clear any pending Halting debug events

bits(64) new_pc;
bits(64) spsr;

if UsingAArch32() then
   new_pc = ZeroExtend(DLR);
   spsr = ZeroExtend(DSPSR);
else
   new_pc = DLR_EL0;
   spsr = DSPSR_EL0;
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
if UsingAAArch32() then
   SetPSTATEFromPSR(spsr<31:0>);                  // Can update privileged bits, even at EL0
else
   SetPSTATEFromPSR(spsr);                        // Can update privileged bits, even at EL0

boolean branch_conditional = FALSE;
if UsingAAArch32() then
   if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
   // AArch32 branch
   BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
else
   // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
   if spsr<4> == '1' & ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
      new_pc<63:32> = Zeros();
   // A type of branch that is never predicted
   BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');        // Atomically signal restarted
UpdateEDSCRFields();                                // Stop signalling PE state
DisableITRAndResumeInstructionPrefetch();

return;
Halt();

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    bits(32) spsr_32;
    bits(64) spsr_64;
    if UsingAArch32() then
        spsr_32 = GetPSRFromPSTATE(DebugState);
    else
        spsr_64 = GetPSRFromPSTATE(DebugState);
    fi

    if (HaveBTIExt() &
        !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive, DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &
        ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
        if UsingAArch32() then
            spsr_32<11:10> = '00';
        else
            spsr_64<11:10> = '00';
        fi
    fi

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr_32;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr_64;
    fi

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
        EDSCR.MA = '0';
    fi

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.TCO is set 1.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;
        PSTATE.TCO = '1';
        PSTATE.BTYPE = '00';
        PSTATE.I = '0';
    fi

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.
    return;
Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';

Library pseudocode for shared/debug/halting/Halted

// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'}); // Halted

Library pseudocode for shared/debug/halting/HaltingAllowed

// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = SecurityStateAtEL(PSTATE.EL);
case ss of
    when SS_NonSecure return ExternalInvasiveDebugEnabled();
    when SS_Secure return ExternalSecureInvasiveDebugEnabled();

Library pseudocode for shared/debug/halting/Restarting

// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001'; // Restarting

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

StopInstructionPrefetchAndEnableITR();
Library pseudocode for shared/debug/halting/UpdateEDSCRFields

// UpdateEDSCRFields()
// ================
// Update EDSCR PE state fields

UpdateEDSCRFields()

if !Halted() then
  EDSCR.EL = '00';
  EDSCR.NS = bit UNKNOWN;
  EDSCR.RW = '1111';
else
  EDSCR.EL = PSTATE.EL;
  ss = SecurityStateAtEL(PSTATE.EL);
  EDSCR.NS = if ss == SS_Secure then '0' else '1';

  bits(4) RW;
  RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
  if PSTATE.EL != EL0 then
    RW<0> = RW<1>;
  else
    RW<0> = if UsingAArch32() then '0' else '1';
    if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
      RW<2> = RW<1>;
    else
      RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
      if !HaveEL(EL3) then
        RW<3> = RW<2>;
      else
        RW<3> = if ELUsingAArch32(EL3) then '0' else '1';
  end

  // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
  if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
  elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
  elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
  EDSCR.RW = RW;
return;

Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)

// Called after an exception entry or exit, that is, such that the Security state
// and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
// is not implemented, this function might also be called at any time.
ss = SecurityStateAtEL(PSTATE.EL);
base = if ss == SS_Secure then 0 else 4;
if HaltingAllowed() then
  boolean halt;
  if HaveExtendedECDebugEvents() then
    exception_exit = !exception_entry;
    increment = 8;
    ctrl = EDECCR<UInt>(PSTATE.EL) + base + increment>:EDECCR<UInt>(PSTATE.EL) + base>;
    case ctrl of
      when '00'  halt = FALSE;
      when '01'  halt = TRUE;
      when '10'  halt = (exception_exit == TRUE);
      when '11'  halt = (exception_entry == TRUE);
    else
      halt = (EDECCR<UInt>(PSTATE.EL) + base) == '1';
  end
  if halt then Halt(DebugHalt_ExceptionCatch);
Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
    end

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((HaveDoPD() && CTIDEVCTL.OSUCE == '1') || (!HaveDoPD() && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
    end

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
    end

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
    end

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
  os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
  if HaltingAllowed() && EDSR.TDA == '1' && os_lock == '0' then
    Halt(DebugHalt_SoftwareAccess);

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
  if HaltingAllowed() then
    Halt(DebugHalt_EDBGRQ);
  else
    // Otherwise the CTI continues to assert the debug request until it is taken.

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall, boolean reset)
  // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
  // or was cancelled by an asynchronous exception.
  //
  // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
  // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
  // address is the instruction following that which generated the exception.
  //
  // "reset" is TRUE if exiting reset state into the highest EL.

  if reset then assert !Halted();       // Cannot come out of reset halted
  active = EDECR.SS == '1' && !Halted();

  if active && reset then               // Coming out of reset with EDECR.SS set
    EDESR.SS = '1';
  elsif active && HaltingAllowed() then
    boolean advance;
    if exception_generated && exception_target == EL3 then
      advance = syscall || ExternalSecureInvasiveDebugEnabled();
    else
      advance = true;
    end
    if advance then EDESR.SS = '1';
  end

  return;
Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

// ExternalDebugInterruptsDisabled()  
// =================================  
// Determine whether EDSCR disables interrupts routed to 'target'.  

boolean ExternalDebugInterruptsDisabled(bits(2) target) 
{  
    boolean int_dis;  
    SecurityState ss = SecurityStateAtEL(target);  
    if Havev8p4Debug() then  
        if EDSCR.INTdis[0] == '1' then  
            case ss of  
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();  
                when SS_Secure int_dis = ExternalSecureInvasiveDebugEnabled();  
            else  
                int_dis = FALSE;  
        else  
            case target of  
                when EL3 int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());  
                when EL2 int_dis = (EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled());  
                when EL1 if ss == SS_Secure then int_dis = (EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled()); else int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());  
        return int_dis;  
    }  
}

Library pseudocode for shared/debug/pmu/GetNumEventCounters

// GetNumEventCounters()  
// =====================  
// Returns the number of event counters implemented. This is indicated to software at the  
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.  

integer GetNumEventCounters()  
{  
    return integer IMPLEMENTATION_DEFINED "Number of event counters";  
}

Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.  

boolean HasElapsed64Cycles();

Library pseudocode for shared/debug/pmu/PMUCountValue

// PMUCountValue()  
// ===============  
// Implements the PMU threshold function, if implemented.  
// Returns the value to increment event counter 'n' by, if the event it is  
// configured to count yields the value 'V' on this cycle.  

integer PMUCountValue(integer n, integer V)  
{  
    if !HavePMUv3TH() then  
        return V;  
    integer T = UInt(PMEVTYPER_EL0[n].TH);  
    case PMEVTYPER_EL0[n].TC of  
        when '000' return (if V != T then V else 0);  
        when '001' return (if V != T then 1 else 0);  
        when '010' return (if V == T then V else 0);  
        when '011' return (if V == T then 1 else 0);  
        when '100' return (if V >= T then V else 0);  
        when '101' return (if V >= T then 1 else 0);  
        when '110' return (if V < T then V else 0);  
        when '111' return (if V < T then 1 else 0);  
    
}
constant integer CYCLE_COUNTER_ID = 31;

// PMUCounterMask()
// ================
// Return bitmask of accessible PMU counters.

bits(32) PMUCounterMask()
    integer n;
    if UsingAArch32() then
        n = AArch32.GetNumEventCountersAccessible();
    else
        n = AArch64.GetNumEventCountersAccessible();
    return '1' : ZeroExtend(Ones(n), 31);

constant bits(16) PMU_EVENT_SW_INCR       = 0x0000<15:0>;
constant bits(16) PMU_EVENT_INST_RETIRED   = 0x0008<15:0>;
constant bits(16) PMU_EVENT_EXC_TAKEN      = 0x0009<15:0>;
constant bits(16) PMU_EVENT_CPU_CYCLES     = 0x0011<15:0>;
constant bits(16) PMU_EVENT_INST_SPEC      = 0x001B<15:0>;
constant bits(16) PMU_EVENT_CHAIN          = 0x001E<15:0>;

// PMUEvent()
// ===========
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) event)
    PMUEvent(event, 1);

// PMUEvent()
// ===========
// Accumulate a PMU Event.

PMUEvent(bits(16) event, integer increment)
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(event, increment, idx);

// PMUEvent()
// ===========
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) event, integer increment, integer idx)
    if !HavePMUv3() then
        return;
    if UsingAArch32() then
        if PMEVTYPER[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
        else
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPER_EL0[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;

array integer PMUEventAccumulator[0..30];  // Accumulates PMU events for a cycle
// CreatePCSample()
// ================
CreatePCSample()
  // In a simple sequential execution of the program, CreatePCSample is executed each time the PE executes an instruction that can be sampled. An implementation is not constrained such that reads of EDPCSRlo return the current values of PC, etc.

  pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
  pc_sample.pc = ThisInstrAddr();
  pc_sample.el = PSTATE.EL;
  pc_sample.rw = if UsingAArch32() then '0' else '1';
  pc_sample.ns = if IsSecure() then '0' else '1';
  pc_sample.contextidr = if ELUsingAArch32_EL1 then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
  pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

  if pc_sample.has_el2 then
    if ELUsingAArch32_EL2 then
      pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
    elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
      pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
      pc_sample.vmid = VTTBR_EL2.VMID;
      if (HaveVirtHostExt() || HaveV82Debug() && !ELUsingAArch32_EL2) then
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
      else
        pc_sample.contextidr_el2 = bits(32) UNKNOWN;
      end
      pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
  end
  return;
// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
  IMPLEMENTATION_DEFINED "generate error response";
  return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

bits(32) sample;
if pc_sample.valid then
  sample = pc_sample.pc<31:0>;
  if update then
    if HaveVirtHostExt() && EDSCR.SC2 == '1' then
      EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
      EDPCSRhi.EL = pc_sample.el;
      EDPCSRhi.NS = pc_sample.ns;
    else
      EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
      EDCIDSR = pc_sample.contextidr;
    end
    EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                else bits(32) UNKNOWN);
    else
      EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} then pc_sample.vmid else
                        Zeros());
      EDVIDSR.NS = pc_sample.ns;
      EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
      EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
    end
    // The conditions for setting HV are not specified if PCSRhi is zero.
    // An example implementation may be "pc_sample.rw".
    EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
  else
    sample = Ones(32);
    if update then
      EDPCSRhi = bits(32) UNKNOWN;
      EDCIDSR = bits(32) UNKNOWN;
      EDVIDSR = bits(32) UNKNOWN;
  end

return sample;

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

type PCSample is (boolean valid,
  bits(64) pc,
  bits(2) el,
  bit rw,
  bit ns,
  boolean has_el2,
  bits(32) contextidr,
  bits(32) contextidr_el2,
  boolean el0h,
  bits(16) vmid)

PCSample pc_sample;
Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

// PMPCSR[] (read)
// ===============

bits(32) PMPCSR[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
  IMPLEMENTATION_DEFINED "generate error response";
  return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

bits(32) sample;
if pc_sample.valid then
  sample = pc_sample.pc<31:0>;
  if update then
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
    PMPCSR.EL = pc_sample.el;
    PMPCSR.NS = pc_sample.ns;
    PMCID1SR = pc_sample.contextidr;
    PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;
    PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1, EL0} && !pc_sample.el0h
       then pc_sample.vmid else bits(16) UNKNOWN);
  else
    sample = Ones(32);
    if update then
      PMPCSR<55:32> = bits(24) UNKNOWN;
      PMPCSR.EL = bits(2) UNKNOWN;
      PMPCSR.NS = bit UNKNOWN;
      PMCID1SR = bits(32) UNKNOWN;
      PMCID2SR = bits(32) UNKNOWN;
      PMVIDSR.VMID = bits(16) UNKNOWN;

return sample;

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

// CheckSoftwareStep()
// ===================

// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

// Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
// AArch32 state. However, because Software Step is only active when the debug target Exception
// level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
step_enabled = !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS == '1';
if step_enabled && PSTATE.SS == '0' then
  AArch64.SoftwareStepException();
Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    if UsingAArch32() then
        assert N == 32;
    else
        assert N == 64;

    assert Halted() || Restarting() || PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    boolean valid;
    bits(2) dest;
    if IllegalExceptionReturn(spsr) then
        dest = PSTATE.EL;
    else
        (valid, dest) = ELFromSPSR(spsr);  assert valid;

    dest_is_secure = IsSecureBelowEL3() || dest == EL3;
    bit mask;
    boolean enabled_at_dest;
    dest_using_32 = (if dest == EL0 then spsr<4> == '1' else ELUsingAArch32(dest));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, dest_is_secure);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, dest_is_secure, mask);

    ELd = DebugTargetFrom(dest_is_secure);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;

Library pseudocode for shared/debug/softwarestep/SSAdvance

// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()
  // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
  // current Software Step state machine. However, this check is made to illustrate that the
  // processor only needs to consider advancing the state machine from the active-not-pending
  // state.

  target = DebugTarget();
  step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
  active_not_pending = step_enabled && PSTATE.SS == '1';

  if active_not_pending then PSTATE.SS = '0';

  return;
Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

// Returns TRUE if the previously executed instruction was executed in the // inactive state, that is, if it was not itself stepped. // Might return TRUE or FALSE if the previously executed instruction was an ISB // or ERET executed in the active-not-pending state, or if another exception // was taken before the Software Step exception. Returns FALSE otherwise, // indicating that the previously executed instruction was executed in the // active-not-pending state, that is, the instruction was stepped.
boolean SoftwareStep_DidNotStep();

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

// Returns a value that describes the previously executed instruction. The // result is valid only if SoftwareStep_DidNotStep() returns FALSE. // Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX // that failed its condition code test. Otherwise returns TRUE if the // instruction was a Load-Exclusive class instruction, and FALSE if the // instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

// ConditionSyndrome() // ============ // Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

bits(5) syndrome;

if UsingAArch32() then
    cond = AArch32.CurrentCond();
    if PSTATE.T == '0' then // A32
        syndrome<4> = '1';
        // A conditional A32 instruction that is known to pass its condition code check // can be presented either with COND set to 0xE, the value for unconditional, or // the COND value held in the instruction.
        if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
            syndrome<3:0> = '1110';
        else
            syndrome<3:0> = cond;
        end;
    else // T32
        // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether: // * CV set to 0 and COND is set to an UNKNOWN value // * CV set to 1 and COND is set to the condition code for the condition that // applied to the instruction.
        if boolean IMPLEMENTATION DEFINED "Condition valid for trapped T32" then
            syndrome<4> = '1';
            syndrome<3:0> = cond;
        else
            syndrome<4> = '0';
            syndrome<3:0> = bits(4) UNKNOWN;
        end;
    end;
else
    syndrome<4> = '1';
    syndrome<3:0> = '1110';
end;
return syndrome;
library pseudocode for shared/exceptions/exceptions/Exception

description

enumeration Exception {Exception_Uncategorized, // Uncategorized or unknown reason
Exception_WFxTrap, // Trapped WFI or WFE instruction
Exception_CPI5RTTrap, // Trapped AArch32 MCR or MRC access, coproc=0b1111
Exception_CPI5RRTTrap, // Trapped AArch32 MRCR or MRRC access, coproc=0b1111
Exception_CPI4RTTrap, // Trapped AArch32 MCR or MRC access, coproc=0b1110
Exception_CPI4DTrap, // Trapped AArch32 LDC or STC access, coproc=0b1110
Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
Exception_FPIDTrap, // Trapped access to SIMD or FP ID register
Exception_LDST64BTrap, // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
// Trapped BXJ instruction not supported in Armv8
Exception_PACTrap, // Trapped invalid PAC use
Exception_IllegalState, // Illegal Execution state
Exception_SupervisorCall, // Supervisor Call
Exception_HypervisorCall, // Hypervisor Call
Exception_MonitorCall, // Monitor Call or Trapped SMC instruction
Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
Exception_ERetTrap, // Trapped invalid ERET use
Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
Exception_PCAlignment, // PC alignment fault
Exception_DataAbort, // Data Abort
Exception_NV2DataAbort, // Data abort at EL1 reported as being from EL2
Exception_PACFail, // PAC Authentication failure
Exception_SPAlignment, // SP alignment fault
Exception_FPTrappedException, // IEEE trapped FP exception
ExceptionSError, // SError interrupt
Exception_Breakpoint, // (Hardware) Breakpoint
Exception_SoftwareStep, // Software Step
Exception_Watchpoint, // Watchpoint
Exception_NV2Watchpoint, // Watchpoint at EL1 reported as being from EL2
Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
Exception_VectorCatch, // AArch32 Vector Catch
Exception_IRQ, // IRQ interrupt
Exception_SVEAccessTrap, // HCPTR trapped access to SVE
Exception_BranchTarget, // Branch Target Identification
Exception_MemCpyMemSet, // Exception from a CPY* or SET* instruction
Exception_FIQ}; // FIQ interrupt

library pseudocode for shared/exceptions/exceptions/ExceptionRecord

description

type ExceptionRecord is (Exception exctype, // Exception class
bits(25) syndrome, // Syndrome record
bits(5) syndrome2, // ST64BV(0) return value register specifier
bits(64) vaddress, // Virtual fault address
boolean ipavalid, // Validity of Intermediate Physical fault address
bit NS, // Intermediate Physical fault address space
bits(52) ipaddress, // Intermediate Physical fault address
boolean trappedsyscallinst) // Trapped SVC or SMC instruction
Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

// ExceptionSyndrome()
// ================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)
{
    ExceptionRecord r;
    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.syndrome2 = Zeros();
    r.ipaddress = Zeros();
    r.NS = '0';
    r.trappedsyscallinst = FALSE;
    return r;
}

Library pseudocode for shared/functions/aborts/EncodeLDFSC

// EncodeLDFSC()
// ==============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)

    if level == -1 then
        assert Have52BitIPAAndPASpaceExt();
    case statuscode of
        when Fault_AddressSize result = '101001';
        when Fault_Translation result = '101011';
        when Fault_SyncExternalOnWalk result = '010011';
        when Fault_SyncParityOnWalk result = '011011'; assert !HaveRASExt();
        otherwise Unreachable();

    return result;

    case statuscode of
        when Fault_AddressSize result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Permission result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal result = '010000';
        when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity result = '011000';
        when Fault_SyncParityOnWalk result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity result = '011001';
        when Fault_AsyncExternal result = '010001';
        when Fault_Alignment result = '100001';
        when Fault_Debug result = '100010';
        when Fault_TLBConflict result = '110000';
        when Fault_HWUpdateAccessFlag result = '110001'; // IMPLEMENTATION DEFINED
        when Fault_Lockdown result = '110100'; // IMPLEMENTATION DEFINED
        when Fault_Exclusive result = '110101'; // IMPLEMENTATION DEFINED
        otherwise Unreachable();

    return result;
Library pseudocode for shared/functions/aborts/IPAValid

// IPAValid()
// =========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    if fault.s2fswalk then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Permission, Fault_Translation, Fault_AddressSize};
    elsif fault.secondstage then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;

Library pseudocode for shared/functions/aborts/IsAsyncAbort

// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsDebugException

// IsDebugException()
// =============

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
Library pseudocode for shared/functions/aborts/IsExternalAbort

// IsExternalAbort()
// ================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
assert statuscode != Fault_None;
return (statuscode IN {
    Fault_SyncExternal,
    Fault_SyncParity,
    Fault_SyncExternalOnWalk,
    Fault_SyncParityOnWalk,
    Fault_AsyncExternal,
    Fault_AsyncParity
});

// IsExternalAbort()
// ================

boolean IsExternalAbort(FaultRecord fault)
return IsExternalAbort(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
assert statuscode != Fault_None;
return (statuscode IN {
    Fault_SyncExternal,
    Fault_SyncParity,
    Fault_SyncExternalOnWalk,
    Fault_SyncParityOnWalk
});

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
return IsExternalSyncAbort(fault.statuscode);
Library pseudocode for shared/functions/aborts/IsFault

// IsFault()
//=========
// Return TRUE if a fault is associated with an address descriptor.

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
//=========
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
//=========
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;

Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

// IsSErrorInterrupt()
//==============
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
//==============

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsSecondStage

// IsSecondStage()
//===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.secondstage;

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is only valid for certain types of access instruction.
// <9:8> - Access size.
// <7>  - Sign extended (for loads).
// <6:2> - Transfer register.
// <1>  - Transfer register is 64-bit.
// <0>  - Instruction has acquire/release semantics.
bits(11) LSInstructionSyndrome();
Library pseudocode for shared/functions/cache/CACHE_OP

// CACHE_OP()
// =========
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/cache/CPASAtPAS

// CPASAtPAS()                      
// =========                              
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS_NonSecure
            return CPAS_NonSecure;
        when PAS_Secure
            return CPAS_Secure;

Library pseudocode for shared/functions/cache/CPASAtSecurityState

// CPASAtSecurityState()            
// ===============                 
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS_NonSecure
            return CPAS_NonSecure;
        when SS_Secure
            return CPAS_SecureNonSecure;

Library pseudocode for shared/functions/cache/CacheOp

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};

Library pseudocode for shared/functions/cache/CacheOpScope

enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};

Library pseudocode for shared/functions/cache/CachePASpace

enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_SecureNonSecure, // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
type CacheRecord is (  
    AccType acctype,           // Access type  
    CacheOp cacheop,           // Cache operation  
    CacheOpScope opscope,      // Cache operation type  
    CacheType cachetype,       // Cache type  
    bits(64) regval,           // For VA operations  
    FullAddress paddress,      // For SW operations  
    bits(64) vaddress,         // For VA operations  
    integer set,               // For SW operations  
    integer way,               // For SW operations  
    integer level,             // For SW operations  
    Shareability shareability, // For cache operations to full cache or by set/way  
    boolean translated,        // For operations by address, PA space in paddress  
    boolean is_vmid_valid,     // is vmid valid for current context  
    bits(16) vmid,             // is vmid valid for current context  
    boolean is_asid_valid,     // is asid valid for current context  
    bits(16) asid,             // is asid valid for current context  
    SecurityState security,    // For cache operations to full cache or by set/way  
    CachePASpace cpas          // For operations by address, PA space in paddress  
)

enumeration CacheType {  
    CacheType_Data,  
    CacheType_Tag,  
    CacheType_Data_Tag,  
    CacheType_Instruction  
};

// DCInstNeedsTranslation()  
// ================  
// Check whether Data Cache operation needs translation.  

boolean DCInstNeedsTranslation(CacheOpScope oopscope)  
if CLIDR_EL1.LoC == '000' then  
    return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoC is before any cache";  
else  
    if CLIDR_EL1.LoUU == '000' && oopscope == CacheOpScope_PoU then  
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoU is before any cache";  
    else  
        return TRUE;  

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)  
    level = UInt(regval[3:1]);  
    (set, way, linesize) = GetCacheInfo(level, cachetype);  
    return (set, way, level);

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

// ICInstNeedsTranslation()
// ========================
// Check whether Instruction Cache operation needs translation.
boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";

Library pseudocode for shared/functions/common/ASR

// ASR()
// =====
bits(N) ASR<bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

Library pseudocode for shared/functions/common/ASR_C

// ASR_C()
// =======
<bits(N), bit) ASR_C<bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

Library pseudocode for shared/functions/common/Abs

// Abs()
// =====
integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====
real Abs(real x)
    return if x >= 0.0 then x else -x;

Library pseudocode for shared/functions/common/Align

// Align()
// ========
integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// ========
<bits(N) Align<bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
Library pseudocode for shared/functions/common/BitCount

// BitCount()
// =========

integer BitCount(bits(N) x)
integer result = 0;
for i = 0 to N-1
    if x<i> == '1' then
        result = result + 1;
return result;

Library pseudocode for shared/functions/common/CountLeadingSignBits

// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);

Library pseudocode for shared/functions/common/CountLeadingZeroBits

// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
return N - (HighestSetBit(x) + 1);

Library pseudocode for shared/functions/common/Elem

// Elem[] - non-assignment form
// ===========================

bits(size) Elem<bits(N) vector, integer e, integer size]
assert e >= 0 && (e+1)*size <= N;
return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// ===========================

bits(size) Elem<bits(N) vector, integer e]
return Elem<vector, e, size>;

// Elem[] - assignment form
// ========================

Elem<bits(N) &vector, integer e, integer size] = bits(size) value
assert e >= 0 && (e+1)*size <= N;
vector<(e+1)*size-1:e*size> = value;
return;

// Elem[] - assignment form
// ========================

Elem<bits(N) &vector, integer e] = bits(size) value
Elem<vector, e, size] = value;
return;

Shared Pseudocode Functions
Library pseudocode for shared/functions/common/Extend

```c
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```

Library pseudocode for shared/functions/common/HighestSetBit

```c
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

Library pseudocode for shared/functions/common/Int

```c
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

Library pseudocode for shared/functions/common/IsOnes

```c
// IsOnes()
// =======

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

Library pseudocode for shared/functions/common/IsZero

```c
// IsZero()
// =======

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

Library pseudocode for shared/functions/common/IsZeroBit

```c
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```
Library pseudocode for shared/functions/common/LSL

// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
assert shift >= 0;
bits(N) result;
if shift == 0 then
result = x;
else
(result, -) = LSL_C(x, shift);
return result;

Library pseudocode for shared/functions/common/LSL_C

// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
assert shift > 0;
extended x = x : Zeros(shift);
result = extended x<N-1:0>;
carry_out = extended x<N>;
return (result, carry_out);

Library pseudocode for shared/functions/common/LSR

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
assert shift >= 0;
bits(N) result;
if shift == 0 then
result = x;
else
(result, -) = LSR_C(x, shift);
return result;

Library pseudocode for shared/functions/common/LSR_C

// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
assert shift > 0;
extended x = ZeroExtend(x, shift+N);
result = extended x<shift+N-1:shift>;
carry_out = extended x<shift-1>;
return (result, carry_out);

Library pseudocode for shared/functions/common/LowestSetBit

// LowestSetBit()
// ================

integer LowestSetBit(bits(N) x)
for i = 0 to N-1
  if x<i> == '1' then return i;
return N;
Library pseudocode for shared/functions/common/Max

```plaintext
// Max()
// =====
integer Max(integer a, integer b)
    return if a >= b then a else b;
// Max()
// =====
real Max(real a, real b)
    return if a >= b then a else b;
```

Library pseudocode for shared/functions/common/Min

```plaintext
// Min()
// =====
integer Min(integer a, integer b)
    return if a <= b then a else b;
// Min()
// =====
real Min(real a, real b)
    return if a <= b then a else b;
```

Library pseudocode for shared/functions/common/Ones

```plaintext
// Ones()
// ======
bits(N) Ones(integer N)
    return Replicate('1',N);
// Ones()
// ======
bits(N) Ones()
    return Ones(N);
```

Library pseudocode for shared/functions/common/ROR

```plaintext
// ROR()
// =====
bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ROR_C

```plaintext
// ROR_C()
// ======
(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```
library functions

// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
   assert N MOD M == 0;
   return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);

// RoundDown()
// ============

integer RoundDown(real x);

// RoundTowardsZero()
// =================

integer RoundTowardsZero(real x)
   return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);

// RoundUp()
// =========

integer RoundUp(real x);

// SInt()
// ======

integer SInt(bits(N) x)
   result = 0;
   for i = 0 to N-1
      if x<i> == '1' then result = result + 2^i;
   if x<N-1> == '1' then result = result - 2*N;
   return result;

// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
   assert N >= M;
   return Replicate(x<M-1>, N-M) : x;

// Split64to32()
// =============

(bits(32), bits(32)) Split64to32(bits(64) value)
   return (value<63:32>, value<31:0>);
Library pseudocode for shared/functions/common/UInt

// UInt()
// ======
integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;

Library pseudocode for shared/functions/common/ZeroExtend

// ZeroExtend()
// ============
bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

Library pseudocode for shared/functions/common/Zeros

// Zeros()
// =======
bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =======
bits(N) Zeros()
    return Zeros(N);
Library pseudocode for shared/functions/counters/AArch32.CheckTimerConditions

// AArch32.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();
    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S, CNTP_CTL_S.IMASK, InterruptID_CNTP);
            CNTP_CTL_S.ISTATUS = if status then '1' else '0';
        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS, CNTP_CTL_NS.IMASK, InterruptID_CNTP);
            CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL, CNTP_CTL.IMASK, InterruptID_CNTP);
            CNTP_CTL.ISTATUS = if status then '1' else '0';
        if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTHP_CVAL, CNTHP_CTL.IMASK, InterruptID_CNTHP);
            CNTHP_CTL.ISTATUS = if status then '1' else '0';
        if CNTV_CTL_EL0.ENABLE == '1' then
            status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0, CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
            CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';
    return;
// AArch64.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
bool status;
bits(64) offset;
bool ecv = FALSE;
if HaveECVExt() then
  ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
if ecv then
  offset = CNTPOFF_EL2;
else
  offset = Zeros(64);
if CNTP_CTL_EL0.ENABLE == '1' then
  status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                               CNTP_CTL_EL0.IMASK, InterruptID_CNTP);
  CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
    CNTHP_CTL_EL2.ENABLE == '1') then
  status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                               CNTHP_CTL_EL2.IMASK, InterruptID_CNTHP);
  CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
if HaveEL(EL2) && HaveSecureEL2Ext() && CNTHPS_CTL_EL2.ENABLE == '1' then
  status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                               CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
  CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';
if CNTPS_CTL_EL1.ENABLE == '1' then
  status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1,
                                CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS);
  CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';
if CNTP_CTL_EL0.ENABLE == '1' then
  status = IsTimerConditionMet(CNTOFF_EL2, CNTP_CVAL_EL0,
                               CNTP_CTL_EL0.IMASK, InterruptID_CNTP);
  CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
    CNTHV_CTL_EL2.ENABLE == '1') then
  status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                               CNTHV_CTL_EL2.IMASK, InterruptID_CNTHV);
  CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';
if ((HaveSecureEL2Ext() && HaveVirtHostExt()) &&
    CNTHVS_CTL_EL2.ENABLE == '1') then
  status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                               CNTHVS_CTL_EL2.IMASK, InterruptID_CNTHVS);
  CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
return;
Library pseudocode for shared/functions/counters/GenericCounterTick

// GenericCounterTick()
// ===============
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
bits(64) prev_physical_count;
if CNTCR.EN == '0' then
  if !HaveAArch64() then
    AArch32.CheckTimerConditions();
  else
    AArch64.CheckTimerConditions();
  return;
prev_physical_count = PhysicalCountInt();
if HaveCNTSCExt() && CNTCR.SCEN == '1' then
  PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR);
else
  PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
if !HaveAArch64() then
  AArch32.CheckTimerConditions();
else
  AArch64.CheckTimerConditions();
  TestEventCNTP(prev_physical_count, PhysicalCountInt());
  TestEventCNTV(prev_physical_count, PhysicalCountInt());
return;

Library pseudocode for shared/functions/counters/IsTimerConditionMet

// IsTimerConditionMet()
// ===============

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value, bits(1) imask, InterruptID intid)

boolean conditon_met;
signal level;
condition_met = (UInt(PhysicalCountInt() - offset) -
                 UInt(compare_value)) >= 0;
level = if condition_met && imask == '0' then HIGH else LOW;
SetInterruptRequestLevel(intid, level);
return condition_met;

Library pseudocode for shared/functions/counters/PhysicalCount

bits(88) PhysicalCount;

Library pseudocode for shared/functions/counters/SetEventRegister

// SetEventRegister()
// ===============
// Sets the Event Register of this PE

SetEventRegister()
EventRegister = '1';
return;
Library pseudocode for shared/functions/counters/TestEventCNTP

    // TestEventCNTP()
    // ===============
    // Generate Event stream from the physical counter
    TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    if CNTHCTL_EL2.EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2.EVNTI);
        if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
            n = n + 8;
        boolean ecv = FALSE;
        if HaveECVExt() then
            ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
                   SCR_EL3.ECVEn == '1');
        offset = if ecv then CNTPOFF_EL2 else Zeros(64);
        samplebit = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTHCTL_EL2.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then
                SetEventRegister();
            else
                if previousbit == '1' && samplebit == '0' then
                    SetEventRegister();
        return;
    Library pseudocode for shared/functions/counters/TestEventCNTV

    // TestEventCNTV()
    // ===============
    // Generate Event stream from the virtual counter
    TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    if (!((HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
           CNTKCTL_EL1.EVNTEN == '1')) then
        n = UInt(CNTKCTL_EL1.EVNTI);
        if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
            n = n + 8;
        if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
            offset = CNTVOFF_EL2;
        else
            offset = Zeros(64);
        samplebit = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTKCTL_EL1.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then
                SetEventRegister();
            else
                if previousbit == '1' && samplebit == '0' then
                    SetEventRegister();
        return;
    Library pseudocode for shared/functions/crc/BitReverse

    // BitReverse()
    // =============
    bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;
Library pseudocode for shared/functions/crc/HaveCRCExt

```c
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```c
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```c
// AESInvMixColumns()
// ==============

// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op< 0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR FFmul09(in3<c*8+:8>);
        out1<c*8+:8> = FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> = FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR FFmul0B(in3<c*8+:8>);
        out3<c*8+:8> = FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR FFmul0E(in3<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> : 
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> : 
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> : 
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```c
// AESInvShiftRows()
// ================

// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return ( 
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> : 
        op< 95: 88> : op<119:112> : op< 15:  8> : op< 39: 32> : 
        op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0> 
    );
```
Library pseudocode for shared/functions/crypto/AESInvSubBytes

// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
// Inverse S-box values
bits(16*16*8) GF2_inv = {
  /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
  /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> : 0x61995383cbbec8b0f52aee4d3be0a0<127:0> :
  /*E*/ 0xef9cc939f7ae52d0d4ab519a97f5160<127:0> : 0x5feca8079f1012b131c707833a8dd1f<127:0> :
  /*D*/ 0xef9cc9939f7ae52d0d4a5b19a97f5160<127:0> :
  /*C*/ 0x5feca8079f1012b131c707833a8dd1f<127:0> :
  /*B*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*A*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*9*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*8*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*7*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*6*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*5*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*4*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*3*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*2*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*1*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0> :
  /*0*/ 0x6edf751ce837ff9e82535ade72274ac96<127:0>
};
bits(128) out;
for i = 0 to 15
  out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
return out;

Library pseudocode for shared/functions/crypto/AESMixColumns

// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns.

bits(128) AESMixColumns(bits(128) op)
bits(4*8) in0 = op<96+:8>: op<64+:8>: op<32+:8>: op<0+:8>;
bits(4*8) in1 = op<104+:8>: op<72+:8>: op<40+:8>: op<8+:8>;
bits(4*8) in2 = op<112+:8>: op<80+:8>: op<48+:8>: op<16+:8>;
bits(4*8) in3 = op<120+:8>: op<88+:8>: op<56+:8>: op<24+:8>;

bits(4*8) out0;
bits(4*8) out1;
bits(4*8) out2;
bits(4*8) out3;
for c = 0 to 3
  out0<c*8+:8> = FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR in2<c*8+:8> EOR in3<c*8+:8> EOR
  out1<c*8+:8> = in0<c*8+:8> EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>) EOR
  out2<c*8+:8> = in0<c*8+:8> EOR in1<c*8+:8> EOR FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>) EOR FFmul03(in3<c*8+:8>) EOR
  out3<c*8+:8> = FFmul03(in0<c*8+:8>) EOR in1<c*8+:8> EOR in2<c*8+:8> EOR FFmul03(in3<c*8+:8>) EOR FFmul03(in3<c*8+:8>) EOR
return {
  out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> : out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> : out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> : out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
};
Library pseudocode for shared/functions/crypto/AESShiftRows

```plaintext
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op<127:120> : op< 87: 80> : op< 47: 40> : op< 7:  0>
    );
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```plaintext
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (/*
        F E D C B A 9 8 7 6 5 4 3 2 1 0
        */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0xe96d186b95736510f6934866b53e70<127:0> :
        /*C*/ 0x848bbd4b1f74de86ca6b3ae12e2578ba<127:0> :
        /*B*/ 0x08ae7a65ae4f566ca94ed5866d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4668902a22dc4f8160<127:0> :
        /*8*/ 0x73195d6437e7c41744975fec130cdd<127:0> :
        /*7*/ 0xd2f3ff1021dad6b6cf5389d928f40a351<127:0> :
        /*6*/ 0xda89f3c50702f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf5f84c4a39be6b6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d3b52a05a6eb1a2c8309<127:0> :
        /*3*/ 0x75b227ebe2b8012029a859618c323c704<127:0> :
        /*2*/ 0x1531d871fe5a534ccf73f362693fd6b7<127:0> :
        /*1*/ 0x6722a49cafa2d4ad50759fa7d9c82ca<127:0> :
        /*0*/ 0x67b6a7b77fe2b670130c56f6bf27b777c63<127:0> 
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt>(op<i*8+:8>)*8+:8>
    return out;
```
Library pseudocode for shared/functions/crypto/FFmul02

```c
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)

bits(256*8) FFmul_02 = ( /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /* F*/ 0x5E7E1E3EDF8F5F7F1F3DFFF9FB<127:0>:
    /* E*/ 0xC5C7C13CDDCFC9CBD5D0D13D0D0F9DB<127:0>:
    /* D*/ 0xA5A7A1A3AFA9AB85B7B18BDBF89BB<127:0>:
    /* C*/ 0x85B781B3D0D8F89BB959791939D9F999B<127:0>:
    /* B*/ 0x656761636D6F696757777137D7F797B<127:0>:
    /* A*/ 0x454741434D4F494B555751535D5F95B<127:0>:
    /* 9*/ 0x25272123222F2928B35373133D0F939B<127:0>:
    /* 8*/ 0x050701000D0F080B15111113111F91B<127:0>:
    /* 7*/ 0xFEFCFAF86F6F20E8ECEAE86E6E2E0<127:0>:
    /* 6*/ 0xDBDCBA806D0D2DCCECCAC864C42C0<127:0>:
    /* 5*/ 0xBEBCBA8B684B820AEACAA86A42A20<127:0>:
    /* 4*/ 0x9E9C9A8969492988888888280<127:0>:
    /* 3*/ 0x7E7C7A7676772706666666260<127:0>:
    /* 2*/ 0x5E5C5A8565452504E44A844A442420<127:0>:
    /* 1*/ 0x3E3C3A3863432320222A2826242220<127:0>:
    /* 0*/ 0x1E1C1A1B161412100E0806040020<127:0> )

return FFmul_02<UInt(b)*8:+8>;
```

Library pseudocode for shared/functions/crypto/FFmul03

```c
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)

bits(256*8) FFmul_03 = ( /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /* F*/ 0x1A191C1F161510130201040D0E0D080B<127:0>:
    /* E*/ 0x2A292C2F26252023323134373E30383B<127:0>:
    /* D*/ 0x7A797F7675736261646766668686<127:0>:
    /* C*/ 0xA494C4F46454352154575E5B588<127:0>:
    /* B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECD2C8C<127:0>:
    /* A*/ 0x8AE9ECEF6E5E0E3F2F1F7FEFDF8FB<127:0>:
    /* 9*/ 0xBAB9BCBF86B5B0B32A1A47EA06A9AB<127:0>:
    /* 8*/ 0xBAB9B9685B0B32A1A47EA06A9AB<127:0>:
    /* 7*/ 0x81B2B7B8DBDEB8899994F9596390<127:0>:
    /* 6*/ 0xB1B2B7B8DBDEB8899994F9596390<127:0>:
    /* 5*/ 0x81B2B7B8DBDEB8899994F9596390<127:0>:
    /* 4*/ 0xD1D2D1D0D0DDB8C9ACFCC5C6C3C0<127:0>:
    /* 3*/ 0x41D247444D4E4B48595AF5C5555350<127:0>:
    /* 2*/ 0x71727777777777777777777777<127:0>:
    /* 1*/ 0x1212227224202E2B2839333A3C333330<127:0>:
    /* 0*/ 0x111217141D1E1B18090A0C05060300<127:0> )

return FFmul_03<UInt(b)*8:+8>;
```
Library pseudocode for shared/functions/crypto/FFmul09

```c
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)

bits(256*8) FFmul_09 = (
    /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /*F*/ 0x464F545D626B70790E071C152A233831<127:0>:
    /*E*/ 0x6D6FC4DF2F6E09E978C85BAB3A8A1<127:0>:
    /*D*/ 0x7D746F6659504842353C272E1118030A<127:0>:
    /*C*/ 0xEDE4FF6C90DBD2A5ACB7BE8188939A<127:0>:
    /*B*/ 0x3039222B141D060F7F716A635C554E47<127:0>:
    /*A*/ 0xA0A9B2BB848D969FE81FAF3CC5DDE7<127:0>:
    /*9*/ 0x080219102F263D34434A515B676E757C<127:0>:
    /*8*/ 0x9B92B9080F6A4D3A51C87F7EESEC<127:0>:
    /*7*/ 0xAAA3B81B8E79C95E2EBF09C6C4D4D<127:0>:
    /*6*/ 0x3A3328211E170C05727B06956F444D<127:0>:
    /*5*/ 0x9198B38B5CA7AE90DBC2FD4EE6E6<127:0>:
    /*4*/ 0x0108131A252C373E49405B626467F6<127:0>:
    /*3*/ 0xDCD55EC7F81EAE394D0868FB089A2AB<127:0>:
    /*2*/ 0x4C455E5768617A304D161F209323B<127:0>:
    /*1*/ 0xE7EEF5FCC3AC1D8A6BDD488B29999<127:0>:
    /*0*/ 0x777E6565C555A41483F362D241B0900<127:0>
);
return FFmul_09<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0B

```c
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)

bits(256*8) FFmul_0B = (
    /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /*F*/ 0xA3A8B5BE8F849992FB90ED67DC1CA<127:0>:
    /*E*/ 0x131050E3F3429224B405D6667C71A<127:0>:
    /*D*/ 0x0D0D3CCE54FFE2F90BB9969DACA7BA81<127:0>:
    /*C*/ 0x6B637E7544F5259303B262D1C170A01<127:0>:
    /*B*/ 0x555E434879726F4806661B0212A373C<127:0>:
    /*A*/ 0xE5EEF3F8C9C2D94B0DB6AAB0919A878C<127:0>:
    /*9*/ 0x2E2538330209141F76706665A514C47<127:0>:
    /*8*/ 0x9E9S8E8B2B99A4AF6C6C600BEAE1FCF7<127:0>:
    /*7*/ 0x545F424978736E5650C71A11202B363D<127:0>:
    /*6*/ 0x4E4E2F2F9C83D0D5BC7AAA19098868<127:0>:
    /*5*/ 0x2F2439320308151E777C16A5B504D46<127:0>:
    /*4*/ 0x9F94898B2B88A5AEC7CDD1DABE90DF6<127:0>:
    /*3*/ 0xA29B48E859893F5AF1CE7D6DC0CB<127:0>:
    /*2*/ 0x1219040F3E3528234A415C5766D7078<127:0>:
    /*1*/ 0xD9D2CFC4F5FEE8BB1A979CADA68B0B<127:0>:
    /*0*/ 0x69627F74454E535813A272C1D160B00<127:0>
);
return FFmul_0B<UInt(b)*8+:8>;
```
Library pseudocode for shared/functions/crypto/FFmul0D

// FFmul0D()
// =========
bits(8) FFmul0D(bits(8) b)
bits(256*8) FFmul_0D = (/* F D C B A 9 8 7 6 5 4 3 2 1 0 */
/* ** F */ 0x979A8D80A3AEB9B4FFF2E57C6D1DC<127:0> : /* E */ 0x474A5D50737E69642F2235381B16010C<127:0> :
/* D */ 0x2C21363B1815020F44495E5370766C61<127:0> :
/* C */ 0xF7E66B8C5CD2DF9998E3A0DBAB7<127:0> :
/* B */ 0xF7E0EDC3D49929F8885A66BCCB1<127:0> :
/* A */ 0x2A27303D1E3049424F58557676C61<127:0> :
/* 9 */ 0x414C5B56A76A760CF9E3845557676C61<127:0> :
/* 8 */ 0x919CBBA76A5A88BF2F94E3EDC0D7DA<127:0> :
/* 7 */ 0x4D057A797463E25283F3211C880B6<127:0> :
/* 6 */ 0x9D907BA9A43B5F8EFE21CCDB6D<127:0> :
/* 5 */ 0xF6FBECE1C2C68593E341E10020C<127:0> :
/* 4 */ 0x262B3C3121F80854E435597A7606D<127:0> :
/* 3 */ 0x202D32119E0348455257C71666B<127:0> :
/* 2 */ 0xF9DFEAE7C496E3D989582B8FCA1B6BB<127:0> :
/* 1 */ 0x9896B18CAFA285BBF3EE9E4C7ADD0<127:0> :
/* 0 */ 0x4B46515CC7F26S5823E39417A0DD0<127:0>
); return FFmul_0D<UInt(b)*8:+8>;

Library pseudocode for shared/functions/crypto/FFmul0E

// FFmul0E()
// =========
bits(8) FFmul0E(bits(8) b)
bits(256*8) FFmul_0E = (/* F D C B A 9 8 7 6 5 4 3 2 1 0 */
/* ** F */ 0x8D83919FB5BBA9A7FDF3E1FC5CB0D97<127:0> : /* E */ 0x6D6371F555B49471D1301F25283937<127:0> :
/* D */ 0x365584A4460727C26283A341E10020C<127:0> :
/* C */ 0x3B6BAA48E8929CC6CD04D661E10020C<127:0> :
/* B */ 0x1F6E332181604A505E4C42686647A<127:0> :
/* A */ 0x3C5ECD2D2F86E4A0BEAC228866949<127:0> :
/* 9 */ 0x5F5E76E9C3CDD018B859799B30AFA1<127:0> :
/* 8 */ 0x15B70923203F31665779535D4F4<127:0> :
/* 7 */ 0x2CCC000DEF4AE8E66BC2A0AE49A896<127:0> :
/* 6 */ 0x2C223E3141A08065C52406464A787<127:0> :
/* 5 */ 0x171908052F21333D67697B755F1434D<127:0> :
/* 4 */ 0xF7F9EBE5CFC1D3D87899B95BF1A3AD<127:0> :
/* 3 */ 0x611FF0D73597541B111F0D32927353B<127:0> :
/* 2 */ 0x81BF90D38BB7834881AC1F8EDE3C87C6D<127:0> :
/* 1 */ 0x88497A8F32D9882C9E90C4D0D68BF0E9E<127:0> :
/* 0 */ 0x5A544648626C7E702A243638121C0E0<127:0>
); return FFmul_0E<UInt(b)*8:+8>;

Library pseudocode for shared/functions/crypto/HaveAESExt

// HaveAESExt()
// ============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```plaintext
// HaveBit128PMULLExt()
// ===============
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```plaintext
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```plaintext
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```plaintext
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```plaintext
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```
Library pseudocode for shared/functions/crypto/HaveSM3Ext

// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";

Library pseudocode for shared/functions/crypto/HaveSM4Ext

// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";

Library pseudocode for shared/functions/crypto/ROL

// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 & shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);

Library pseudocode for shared/functions/crypto/SHA256hash

// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
        maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
        t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
        x<127:96> = t + x<127:96>;
        y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
        <y, x> = ROL(y : x, 32);
    return (if part1 then x else y);

Library pseudocode for shared/functions/crypto/SHAchoose

// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);

Library pseudocode for shared/functions/crypto/SHA256hash
Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

// SHAhashSIGMA0()
// ===============
bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

// SHAhashSIGMA1()
// ===============
bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);

Library pseudocode for shared/functions/crypto/SHAmajority

// SHAmajority()
// =============
bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));

Library pseudocode for shared/functions/crypto/SHAparity

// SHAparity()
// ===========
bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);

Library pseudocode for shared/functions/crypto/Sbox

// Sbox()
// ======
// Used in SM4E crypto instruction
bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fece13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e
    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.
ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/ProcessorID

// Return the ID of the currently executing PE.
integer ProcessorID();

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

// AArch32.HaveHPDExt()
// ====================
boolean AArch32.HaveHPDExt()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 hierarchical permission disables");

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

// AArch64.HaveHPDExt()
// ====================
boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
{
    return (HasArchVersion(ARMv8p1) && HaveEL(EL2) &&
            boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID");
}

// Have52BitIPAAndPASpaceExt()
// ===========================
// Returns TRUE if 52-bit IPA and PA extension support is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
{
    return (HasArchVersion(ARMv8p7) &&
            boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
            Have52BitVAExt() && Have52BitPAExt());
}

// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
{
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support");
}

// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
{
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support");
}

// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
{
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension");
}

// HaveAArch32Int8MatMulExt()
// ==========================// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
{
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension");
}
Library pseudocode for shared/functions/extension/HaveAltFP

// HaveAltFP()
// ===========
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.
boolean HaveAltFP()
    return HasArchVersion(ARMv8p7);

Library pseudocode for shared/functions/extension/HaveAtomicExt

// HaveAtomicExt()
// ===============
boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/HaveBF16Ext

// HaveBF16Ext()
// =============
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.
boolean HaveBF16Ext()
    return (HasArchVersion(ARMv8p6) ||
            (HasArchVersion(ARMv8p2) &&
             boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension");

Library pseudocode for shared/functions/extension/HaveBTIExt

// HaveBTIExt()
// ============
// Returns TRUE if support for Branch Target Indentification is implemented.
boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);

Library pseudocode for shared/functions/extension/HaveBlockBBM

// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requiring
// break-before-make is implemented.
boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);

Library pseudocode for shared/functions/extension/HaveCNTSCExt

// HaveCNTSCExt()
// ===============
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.
boolean HaveCNTSCExt()
    return (HasArchVersion(ARMv8p4) &&
            boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support");
// HaveCommonNotPrivateTransExt()
// ==============================
boolean HaveCommonNotPrivateTransExt()
return HasArchVersion(ARMv8p2);

// HaveDGHExt()
// ============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and
// FALSE otherwise.
boolean HaveDGHExt()
return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";

// HaveDITExt()
// ============
boolean HaveDITExt()
return HasArchVersion(ARMv8p4);

// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.
boolean HaveDOTPExt()
return (HasArchVersion(ARMv8p4) ||
(HasArchVersion(ARMv8p2) &&
boolean IMPLEMENTATION_DEFINED "Has Dot Product extension"));

// HaveDoPD()
// ===========
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.
boolean HaveDoPD()
return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";

// HaveDoubleFaultExt()
// ====================
boolean HaveDoubleFaultExt()
return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESR());

// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.
boolean HaveDoubleLock()
return (!HasArchVersion(ARMv8p4) ||
boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented");
// HaveE0PDEExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.
boolean HaveE0PDEExt()
return HasArchVersion(ARMv8p5);

// HaveECVExt()
// ============
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.
boolean HaveECVExt()
return HasArchVersion(ARMv8p6);

// HaveEMPAMExt()
// ==============
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.
boolean HaveEMPAMExt()
return (HasArchVersion(ARMv8p6) && HaveMPAMExt()) &&
boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension";

// HaveExtendedCacheSets()
// =======================
boolean HaveExtendedCacheSets()
return HasArchVersion(ARMv8p3);

// HaveExtendedECDebugEvents()
// ===========================
boolean HaveExtendedECDebugEvents()
return HasArchVersion(ARMv8p2);

// HaveExtendedExecuteNeverExt()
// =============================
boolean HaveExtendedExecuteNeverExt()
return HasArchVersion(ARMv8p2);

// HaveFCADDExt()
// ==============
boolean HaveFCADDExt()
return HasArchVersion(ARMv8p3);
Library pseudocode for shared/functions/extension/HaveFGTExt

// HaveFGTExt()
// ============
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.
boolean HaveFGTExt()
return HasArchVersion(ARMv8p6);

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

// HaveFJCVTZSExt()
// ================
boolean HaveFJCVTZSExt()
return HasArchVersion(ARMv8p3);

Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

// HaveFP16MulNoRoundingToFP32Ext()
// ===============================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions, and FALSE otherwise
boolean HaveFP16MulNoRoundingToFP32Ext()
if !HaveFP16Ext() then return FALSE;
if HasArchVersion(ARMv8p4) then return TRUE;
return (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");

Library pseudocode for shared/functions/extension/HaveFeatCMOW

// HaveFeatCMOW()
// ==============
// Returns TRUE if the SCTLR_EL1.CMOW bit is implemented and the SCTLR_EL2.CMOW and HCRX_EL2.CMOW bits are implemented if EL2 is implemented.
boolean HaveFeatCMOW()
return HasArchVersion(ARMv8p8);

Library pseudocode for shared/functions/extension/HaveFeatHBC

// HaveFeatHBC()
// =============
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.
boolean HaveFeatHBC()
return HasArchVersion(ARMv8p8);

Library pseudocode for shared/functions/extension/HaveFeatHCX

// HaveFeatHCX()
// =============
// Returns TRUE if HCRX_EL2 Trap Control register is implemented, and FALSE otherwise.
boolean HaveFeatHCX()
return HasArchVersion(ARMv8p7);
// HaveFeatHPMN0()
// ===============
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHPMN0()
    return HasArchVersion(ARMv8p8) && HavePMUv3() && HaveFGEx() && HaveEL(EL2);

// HaveFeatLS64()
// =============
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return (HasArchVersion(ARMv8p7) &&
            boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");

// HaveFeatLS64_ACCDATA()
// ======================
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
    return (HasArchVersion(ARMv8p7) && HaveFeatLS64_V() &&
            boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte EL0 with return instruction support");

// HaveFeatLS64_V()
// ================
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_V()
    return (HasArchVersion(ARMv8p7) && HaveFeatLS64() &&
            boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte with return instruction support");

// HaveFeatMOPS()
// ==============
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatMOPS()
    return HasArchVersion(ARMv8p8);

// HaveFeatNMI()
// =============
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.

boolean HaveFeatNMI()
    return HasArchVersion(ARMv8p8);
// HaveFeatRPRES()
// ===============
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatRPRES()
return HasArchVersion(ARMv8p7) &
(boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision support") 
HaveAltFP();

// HaveFeatTIDCP1()
// ================
// Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
// is implemented if EL2 is implemented.

boolean HaveFeatTIDCP1()
return HasArchVersion(ARMv8p8);

// HaveFeatWFxT()
// ==============
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
return HasArchVersion(ARMv8p7);

// HaveFeatWFxT2()
// ===============
// Returns TRUE if the register number is reported in the ESR_ELx
// on exceptions to WFIT and WFET.

boolean HaveFeatWFxT2()
return HaveFeatWFxT() &
boolean IMPLEMENTATION_DEFINED "Has feature WFxT2";

// HaveFeatXS()
// ============
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
return HasArchVersion(ARMv8p7);

// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
return HasArchVersion(ARMv8p5);
library pseudocode for shared/functions/extension/HaveFlagManipulateExt

// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);

library pseudocode for shared/functions/extension/HaveFrintExt

// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);

library pseudocode for shared/functions/extension/HaveHPMDExt

// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HavePMUv3p1();

library pseudocode for shared/functions/extension/HaveIDSExt

// HaveIDSExt()
// ============
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return HasArchVersion(ARMv8p4);

library pseudocode for shared/functions/extension/HaveIESB

// HaveIESB()
// ===========

boolean HaveIESB()
    return (HaveBASEt() &&
        boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");

library pseudocode for shared/functions/extension/HaveInt8MatMulExt

// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return (HasArchVersion(ARMv8p6) ||
        (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension");

library pseudocode for shared/functions/extension/HaveLSE2Ext

// HaveLSE2Ext()
// =============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return HasArchVersion(ARMv8p4);
Library pseudocode for shared/functions/extension/HaveMPAMExt

// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");

Library pseudocode for shared/functions/extension/HaveMTE2Ext

// HaveMTE2Ext()
// =============
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";

Library pseudocode for shared/functions/extension/HaveMTE3Ext

// HaveMTE3Ext()
// =============
// Returns TRUE if MTE Asymmetric Fault Handling support is implemented, and FALSE otherwise.

boolean HaveMTE3Ext()
    return (HasArchVersion(ARMv8p7) && HaveMTE2Ext()) || (HasArchVersion(ARMv8p5) &&
            boolean IMPLEMENTATION_DEFINED "Has MTE3 extension");

Library pseudocode for shared/functions/extension/HaveMTEEExt

// HaveMTEEExt()
// =============
// Returns TRUE if MTE implemented, and FALSE otherwise.

boolean HaveMTEEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    if HaveMTE2Ext() then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";

Library pseudocode for shared/functions/extension/HaveNV2Ext

// HaveNV2Ext()
// ===========
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return (HasArchVersion(ARMv8p4) && HaveNVExt() &&
            boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");

Library pseudocode for shared/functions/extension/HaveNVExt

// HaveNVExt()
// ===========
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
    return (HasArchVersion(ARMv8p3) &&
            boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization");
// HaveNoSecurePMUDisableOverride()
// ================================
boolean HaveNoSecurePMUDisableOverride() { return HasArchVersion(ARMv8p2); }

// HaveNoninvasiveDebugAuth()
// ==========================
// Returns TRUE if the Non-invasive debug controls are implemented.
boolean HaveNoninvasiveDebugAuth() { return !HasArchVersion(ARMv8p4); }

// HavePAN3Ext()
// =============
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
// and FALSE otherwise.
boolean HavePAN3Ext() { return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) && boolean IMPLEMENTATION_DEFINED "Has PAN3 extension"); }

// HavePANExt()
// ============
boolean HavePANExt() { return HasArchVersion(ARMv8p1); }

// HavePMUv3()
// ===========
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.
boolean HavePMUv3() { return boolean IMPLEMENTATION_DEFINED "Has Performance Monitors extension"; }

// HavePMUv3TH()
// =============
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.
boolean HavePMUv3TH() { return (HasArchVersion(ARMv8p8) && HavePMUv3()) && boolean IMPLEMENTATION_DEFINED "Has PMUv3 threshold extension"); }

// HavePMUv3p1()
// ==============
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.
boolean HavePMUv3p1() { return HasArchVersion(ARMv8p1) && HavePMUv3(); }
// HavePMUv3p4()
// =============
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.
boolean HavePMUv3p4()
    return HasArchVersion(ARMv8p4) && HavePMUv3();

// HavePMUv3p5()
// =============
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.
boolean HavePMUv3p5()
    return HasArchVersion(ARMv8p5) && HavePMUv3();

// HavePMUv3p7()
// =============
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.
boolean HavePMUv3p7()
    return HasArchVersion(ARMv8p7) && HavePMUv3();

// HavePageBasedHardwareAttributes()
// =================================
boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);

// HavePrivATExt()
// ===============
boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);

// HaveQRDMLAHExt()
// ================
boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);

// HaveRASExt()
// ============
boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) || IMPLEMENTATION_DEFINED "Has RAS extension");
Library pseudocode for shared/functions/extension/HaveRNG

```
// HaveRNG()
// =========
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG() {
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";
}
```

Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt() {
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
}
```

Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt() {
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
}
```

Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext() {
    return HasArchVersion(ARMv8p4);
}
```

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
//========================= // Returns TRUE if support for Secure and Non-secure views of debug peripherals // is implemented.

boolean HaveSecureExtDebugView() {
    return HasArchVersion(ARMv8p4);
}
```

Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====================

boolean HaveSelfHostedTrace() {
    return HasArchVersion(ARMv8p4);
}
```

Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

```
// HaveSmallTranslationTblExt()
// =========================== // Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt() {
    return (HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension");
}
```
// HaveSoftwareLock()
// ===============
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
{
    if(Havev8p4Debug())
        return FALSE;
    if(HaveDoPD() && component != Component_CTI)
        return FALSE;
    case component of
    when Component_Debug
        return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
    when Component_PMU
        return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
    when Component_CTI
        return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
    otherwise
        Unreachable();
}

// HaveStage2MemAttrControl()
// =========================
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented.

boolean HaveStage2MemAttrControl()
{
    return HasArchVersion(ARMv8p4);
}

// HaveStatisticalProfiling()
// ==========================
// Returns TRUE if Statistical Profiling Extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfiling()
{
    return HasArchVersion(ARMv8p2);
}

// HaveStatisticalProfilingv1p1()
// =============================
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
{
    return (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
}

// HaveStatisticalProfilingv1p2()
// =============================
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
{
    return (HasArchVersion(ARMv8p7) && HaveStatisticalProfiling() && boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");
}
Library pseudocode for shared/functions/extension/HaveTWEDExt

```plaintext
// HaveTWEDExt()
// =============
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.

boolean HaveTWEDExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";
```

Library pseudocode for shared/functions/extension/HaveTraceExt

```plaintext
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```plaintext
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```plaintext
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveV82Debug

```plaintext
// HaveV82Debug()
// ===============

boolean HaveV82Debug()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```plaintext
// HaveVirtHostExt()
// ================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Havev8p4Debug

```plaintext
// Havev8p4Debug()
// ===============
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean Havev8p4Debug()
    return HasArchVersion(ARMv8p4);
```
Library pseudocode for shared/functions/extension/InsertIESBBeforeException

// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable SError interrupt must be taken before executing any instructions in the exception handler. However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el);

Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite, AddressDescriptor memaddrdesc, integer size, AccessDescriptor accdesc)

assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} || (!HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity, Fault_AsyncParity}));

fault = NoFault();

fault.statuscode = memretstatus.statuscode;
fault.write = iswrite;
fault.extflag = memretstatus.extflag;
fault.acctype = memretstatus.acctype;

// It is implementation specific whether external aborts signaled in-band synchronously are taken synchronously or asynchronously
if (IsExternalSyncAbort(fault) && !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc, size, accdesc)) then
    if fault.statuscode == Fault_SyncParity then
        fault.statuscode = Fault_AsyncParity;
    else
        fault.statuscode = Fault_AsyncExternal;
else
    fault.errortype = PEErrorState(memretstatus);

if IsExternalSyncAbort(fault) then
    if UsingAArch32() then
        AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
    else
        AArch64.Abort(memaddrdesc.vaddress, fault);
else
    PendSErrorInterrupt(fault);

Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc, integer size, AccessDescriptor accdesc)

iswrite = FALSE;
HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

// HandleExternalTTWAbort()
// ================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite, AddressDescriptor memaddrdesc, AccessDescriptor accdesc, integer size, FaultRecord input_fault)

output_fault = input_fault;
output_fault.extflag = memretstatus.extflag;
output_fault.statuscode = memretstatus.statuscode;
if (IsExternalSyncAbort(output_fault) && !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc, size, accdesc)) then
    if output_fault.statuscode == Fault_SyncParity then
        output_fault.statuscode = Fault_AsyncParity;
    else
        output_fault.statuscode = Fault_AsyncExternal;
else
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    else
        output_fault.errortype = PEErorrState(memretstatus);
        if HaveRASExt() then
            output_fault.errortype = bits(2) UNKNOWN;
        if !IsExternalSyncAbort(output_fault) then
            PendSErrorInterrupt(output_fault);
output_fault.statuscode = Fault_None;
return output_fault;

Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

// HandleExternalWriteAbort()
// ================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc, integer size, AccessDescriptor accdesc)

iswrite = TRUE;
HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.

// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

boolean IsExternalAbortTakenSynchronously(
    PhysMemRetStatus memstatus,
    boolean iswrite,
    AddressDescriptor desc,
    integer size,
    AccessDescriptor accdesc);

constant bits(2) Sync_UC   = '10'; // Synchronous Uncontainable
constant bits(2) Sync_UER  = '00'; // Synchronous Recoverable
constant bits(2) Sync_UEO  = '11'; // Synchronous Restartable
constant bits(2) ASync_UC  = '00'; // Asynchronous Uncontainable
constant bits(2) ASync_UER = '01'; // Asynchronous Unrecoverable
constant bits(2) ASync_UEO = '11'; // Asynchronous Restartable
constant bits(2) ASync_UEU = '10'; // Asynchronous Recoverable

bits(2) PEErrorState(PhysMemRetStatus memstatus);

// Pend the SError.
PendSErrorInterrupt(FaultRecord fault);
Library pseudocode for shared/functions/float/bfloat/BFAdd

// BFAdd()
// =========
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)

bits(32) result;

FPCRType fpcr = FPCR[];
(type1,sign1,value1) = BFUnpack(op1);
(type2,sign2,value2) = BFUnpack(op2);
if type1 == FPType_QNaN || type2 == FPType_QNaN then
  result = FPDefaultNaN(fpcr);
else
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if inf1 && inf2 && sign1 == NOT(sign2) then
    result = FPDefaultNaN(fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
    result = FPInfinity('0');
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
    result = FPInfinity('1');
  elsif zero1 && zero2 && sign1 == sign2 then
    result = FPZero(sign1);
  else
    result_value = value1 + value2;
    if result_value == 0.0 then
      result = FPZero('0');    // Positive sign when Round to Odd
    else
      result = BFRound(result_value);
  end
return result;

Library pseudocode for shared/functions/float/bfloat/BFDotAdd

// BFDotAdd()
// =========
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
  bits(16) op2_a, bits(16) op2_b, FPCRType fpcr_in)

FPCRType fpcr = fpcr_in;

bits(32) prod;

prod = BFAdd(BFMul(op1_a, op2_a), BFMul(op1_b, op2_b));
result = BFAdd(addend, prod);

return result;
**Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd**

```c
// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;
    bits(N) result;
    bits(32) sum;
    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
            Elem[result, 2*i + j, 32] = sum;
    return result;
```

---

**Library pseudocode for shared/functions/float/bfloat/BFMul**

```c
// BFMul()
// ========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)

    bits(32) result;
    FPCRTYPE fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPTYPE_QNaN || type2 == FPTYPE_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FPTYPE_Infinity);
        inf2 = (type2 == FPTYPE_Infinity);
        zero1 = (type1 == FPTYPE_Zero);
        zero2 = (type2 == FPTYPE_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);
    return result;
```
Library pseudocode for shared/functions/float/bfloat/BFMulAdd

```c
// BFMulAdd()
// =========
// Used by BFMLALB and BFMLALT instructions.

bits(N) BF MulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean altfp = HaveAltFP() && fpcr.AH == '1'; // When TRUE:
    boolean fpexc = !altfp;                         //     Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             //     Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';             //     Use RNE rounding mode
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFNeg

```c
// BFNeg()
// ======

bits(16) BFNeg(bits(16) op)
    return NOT(op<15>) : op<14:0>;
```
Library pseudocode for shared/functions/float/bfloat/BFRound

// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)

assert op != 0.0;
bits(32) result;

// Format parameters - minimum exponent, numbers of exponent and fraction bits.
minimum_exp = -126;  E = 8;  F = 23;

// Split value into sign, unrounded mantissa and exponent.
bit sign;
real mantissa;
if op < 0.0 then
  sign = '1';  mantissa = -op;
else
  sign = '0';  mantissa = op;

exponent = 0;
while mantissa < 1.0 do
  mantissa = mantissa * 2.0;  exponent = exponent - 1;
while mantissa >= 2.0 do
  mantissa = mantissa / 2.0;  exponent = exponent + 1;

// Fixed Flush-to-zero.
if exponent < minimum_exp then
  return FPZero(sign);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biasied_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Round to Odd
if error != 0.0 then
  int_mant<0> = '1';

// Deal with overflow and generate result.
if biased_exp >= 2^E - 1 then
  result = FPInfinity(sign);  // Overflows generate appropriately-signed Infinity
else
  result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

return result;
BFUnpack()

Unpacks a BFloat16 or single-precision value into its type, sign bit and real number that it represents.
The real number result has the correct sign for numbers and infinities, is very large in magnitude for infinities, and is 0.0 for NaNs.
(These values are chosen to simplify the description of comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)

assert N IN {16, 32};

bit sign;
bits(8) exp;
bits(23) frac;
if N == 16 then
    sign = fpval<15>;
    exp = fpval<14:7>;
    frac = fpval<6:0> : Zeros(16);
else // N == 32
    sign = fpval<31>;
    exp = fpval<30:23>;
    frac = fpval<22:0>;

FPType fptype;
real value;
if IsZero(exp) then
    fptype = FPType_Zero; value = 0.0; // Fixed Flush to Zero
elsif IsOnes(exp) then
    if IsZero(frac) then
        fptype = FPType_Infinity; value = 2.0^1000000;
    else // no SNaN for BF16 arithmetic
        fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^((UInt)(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
    if sign == '1' then value = -value;
return (fptype, sign, value);
Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```c
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr_in, FPRounding rounding_in)
```

```c
FPCRType fpcr = fpcr_in;
FPRounding rounding = rounding_in;

bits(32) result;                                // BF16 value in top 16 bits
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp;                         // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
if altfp then rounding = FPRounding_TIEEVEN;    // Use RNE rounding mode

// Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    if fpcr.DN == '1' then
        result = FPDefaultNaN(fpcr);
    else
        result = FPConvertNaN(op);
elseif fptype == FPType_Infinity then
    result = FPInfinity(sign);
elseif fptype == FPType_Zero then
    result = FPZero(sign);
else
    result = FPRoundCVBF(value, fpcr, rounding, fpexc);

// Returns correctly rounded BF16 value from top 16 bits
return result<31:16>;
```

Library pseudocode for shared/functions/float/bfloat/FPRoundCVBF

```c
// FPRoundCVBF()
// =============
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) FPRoundCVBF(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc)
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);
```
Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```cpp
// FixedToFP()
// ===========
// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and Rounding.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

Library pseudocode for shared/functions/float/fpabs/FPAbs

```cpp
// FPAbs()
// ========

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (ftype, -, -) = FPUnpack(op, fpcr, FALSE);
            if ftype IN {FPType_SNaN, FPType_QNaN} then
                return op; // When fpcr.AH=1, sign of NaN has no consequence
        end

    return '0' : op<N-2:0>;
```
Library pseudocode for shared/functions/float/fpadd/FPAdd

// FPAdd()
// ========
bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
  boolean fpexc = TRUE; // Generate floating-point exceptions
  return FPAdd(op1, op2, fpcr, fpexc);
// FPAdd()
// ========

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

  assert N IN {16,32,64};
  rounding = FPRoundingMode(fpcr);

  (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
  (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

  boolean altfmaxfmin = FALSE; // Do not use altfp mode for FMIN, FMAX and variants
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);
  if !done then
    inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
    if inf1 && inf2 && sign1 == NOT(sign2) then
      result = FPDefaultNaN(fpcr);
    elseif fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
      result = FPInfinity('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
      result = FPInfinity('1');
    elseif zero1 && zero2 && sign1 == sign2 then
      result = FPZero(sign1);
    else
      result_value = value1 + value2;
      if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
      else
        result = FPRound(result_value, fpcr, rounding, fpexc);
      if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
      return result;
Library pseudocode for shared/functions/float/fpcompare/FPCompare

// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

bits(4) result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = '0011';
else if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    if value1 == value2 then
        result = '0110';
    elsif value1 < value2 then
        result = '1000';
    else // value1 > value2
        result = '0010';
        FPProcessDenorms(type1, type2, N, fpcr);

return result;

Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

boolean result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = FALSE;
else if type1 == FPType_SNaN || type2 == FPType_SNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

return result;
Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

boolean result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = FALSE;
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 >= value2);
    FPProcessDenorms(type1, type2, N, fpcr);

return result;

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

boolean result;
if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
    result = FALSE;
    FPProcessException(FPExc_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    result = (value1 > value2);
    FPProcessDenorms(type1, type2, N, fpcr);

return result;
Library pseudocode for shared/functions/float/fpconvert/FPConvert

```c
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)

assert M IN {16,32,64};
assert N IN {16,32,64};
bits(M) result;

// Unpack floating-point operand optionally with flush-to-zero.
(fptype,sign,value) = FPUnpackCV(op, fpcr);

alt_hp = (M == 16) && (fpcr.AHP == '1');

if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  if alt_hp then
    result = FPZero(sign);
  elsif fpcr.DN == '1' then
    result = FPDefaultNaN(fpcr);
  else
    result = FPConvertNaN(op);
    if fptype == FPTYPE_SNaN || alt_hp then
      FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPTYPE_Infinity then
      if alt_hp then
        result = sign:Ones(M-1);
      else
        result = FPInfinity(sign);
      end
      FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPTYPE_Zero then
      result = FPZero(sign);
    else
      result = FPRoundCV(value, fpcr, rounding);
      FPProcessDenorm(fptype, N, fpcr);
  end
else
  result = FPRoundCV(value, fpcr, rounding);
  FPProcessDenorm(fptype, N, fpcr);
end

return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```
Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)

assert N IN {16,32,64};
assert M IN {16,32,64};
bits(M) result;
bits(51) frac;

sign = op<N-1>;

// Unpack payload from input NaN
case N of
    when 64 frac = op<50:0>:
    when 32 frac = op<21:0>: Zeros(29):
    when 16 frac = op<8:0>: Zeros(42):

// Repack payload into output NaN, while
// converting an SNaN to a QNaN.
case M of
    when 64 result = sign: Ones(M-52):frac:
    when 32 result = sign: Ones(M-23):frac<50:29>:
    when 16 result = sign: Ones(M-10):frac<50:42>:

return result;

Library pseudocode for shared/functions/float/fpcrtype/FPCRType

type FPCRType;

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

// FPDecodeRM()
// ============
// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

FPRounding result;
case rm of
    when '00' result = FPRounding_TIEAWAY; // A
    when '01' result = FPRounding_TIEEVEN; // N
    when '10' result = FPRounding_POSINF; // P
    when '11' result = FPRounding_NEGINF; // M

return result;

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

// FPDecodeRounding()
// ==================
// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)

case rmode of
    when '00' return FPRounding_TIEEVEN; // N
    when '01' return FPRounding_POSINF; // P
    when '10' return FPRounding_NEGINF; // M
    when '11' return FPRounding_ZERO; // Z
Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()

    FPCRType fp = FPCR[];
    return FPDefaultNaN(fp);

bits(N) FPDefaultNaN(FPCRType fp)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bit sign = if HaveAltFP() && !UsingAArch32() then fp.AH else '0';

    bits(E) exp  = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;

Library pseudocode for shared/functions/float/fpdiv/FPDiv

// FPDiv()  
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fp)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fp);
    (type2,sign2,value2) = FPUnpack(op2, fp);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fp);

    if !done then
        inf1  = type1 == FPType_Infinity;
        inf2  = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fp);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then
                FPProcessException(FPExc_DivideByZero, fp);
            elsif zero1 || inf2 then
                result = FPZero(sign1 EOR sign2);
            else
                result = FPRound(value1/value2, fp);
        else
            FPProcessDenorms(type1, type2, N, fp);

        return result;

Library pseudocode for shared/functions/float/fpexc/FPExc

enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow, FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```pseudocode
// FPInfinity()
// ============
bits(N) FPInfinity(bit sign)

assert N IN {16, 32, 64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
bits(E) exp = Ones(E);
bits(F) frac = Zeros(F);

return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```pseudocode
// FPMatMulAdd()
// =============
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)

assert N == esize * 2 * 2;
bits(N) result;
bits(esize) prod0, prod1, sum;

for i = 0 to 1 
    for j = 0 to 1 
        sum   = Elem[addend, 2*i + j, esize];
        prod0 = FPMul( Elem[op1, 2*i + 0, esize],
                        Elem[op2, 2*j + 0, esize], fpcr);
        prod1 = FPMul( Elem[op1, 2*i + 1, esize],
                        Elem[op2, 2*j + 1, esize], fpcr);
        sum   = FPAdd( sum, FPAdd(prod0, prod1, fpcr), fpcr);
        Elem[result, 2*i + j, esize] = sum;

return result;
```
Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// ========
bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// ========
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.
bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)
    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if (altfp && type1 == FPTYPE_ZERO & type2 == FPTYPE_ZERO &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE_INFINITY then
            result = FPInfinity(sign);
        elsif fptype == FPTYPE_ZERO then
            sign = sign1 AND sign2;         // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then
                // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            result = FPRound(value, fpcr, rounding, TRUE);
        FPProcessDenorms(type1, type2, N, fpcr);
        return result;
    Shared Pseudocode Functions
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============
bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```
Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

// FPMaxNum()
// =========

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRTypetypetr fpcr)

  assert N IN {16,32,64};
  bits(N) op1 = op1_in;
  bits(N) op2 = op2_in;
  (type1,-,-) = FPUnpack(op1, fpcr);
  (type2,-,-) = FPUnpack(op2, fpcr);

  boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
  boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

  if !(altfp && type1_nan && type2_nan) then
    // Treat a single quiet-NaN as -Infinity.
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
      op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
      op2 = FPInfinity('1');
    altfmaxfmin = FALSE;   // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);
  return result;

Library pseudocode for shared/functions/float/fpmerge/IsMerging

// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRTypetypetr fpcr)
  boolean merge = HaveAltFP() && !UsingAArch32() && fpcr.NEP == '1';
  return merge;
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// =======

// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)
    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if (altfp && type1 == FPType_Zero && type2 == FPType_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);
    if !done then
        FType fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';
            result = FPRound(value, fpcr, rounding, TRUE);
        
        FPProcessDenorms(type1, type2, N, fpcr);
    
    return result;
// FPMinNum()
// =========

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

assert N IN {16,32,64};
bits(N) op1 = op1_in;
bits(N) op2 = op2_in;
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = HaveAltFP() && ! UsingAArch32() && fpcr.AH == '1';

if !(altfp && type1_nan && type2_nan) then
  // Treat a single quiet-NaN as +Infinity.
  if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('0');
  elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('0');
  altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
  result = FPMin(op1, op2, fpcr, altfmaxfmin);
return result;

// FPMul()
// ========

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);
  if (inf1 && zero2) || (zero1 && inf2) then
    result = FPDefaultNaN(fpcr);
  FProcessException(FPExc_InvalidOp, fpcr);
  elsif inf1 || inf2 then
    result = FPInfinity(sign1 EOR sign2);
  elsif zero1 || zero2 then
    result = FPZero(sign1 EOR sign2);
  else
    result = FPRound(value1*value2, fpcr);
  FPProcessDenorms(type1, type2, N, fpcr);
return result;
Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd
// FPMulAdd()
// =========

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)
boolean fpexc = TRUE; // Generate floating-point exceptions
return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =========

// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)
assert N IN {16,32,64};

(typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
rounding = FPRoundingMode(fpcr);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

(done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);
if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
  if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN(fpcr);
  if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
endif
if !done then
  infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
  // Determine sign and type product will have if it does not cause an
  // Invalid Operation.
  signP = sign1 EOR sign2;
  infP  = inf1 || inf2;
  zeroP = zero1 || zero2;
  // Non SNaN-generated Invalid Operation cases are multiples of zero
  // by infinity and additions of opposite-signed infinities.
  invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
  if invalidop then
    result = FPDefaultNaN(fpcr);
  if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (infA && signA == '0') || (infP && signP == '0') then
    result = FPIInfinity('0');
  elsif (infA && signA == '1') || (infP && signP == '1') then
    result = FPIInfinity('1');
  endif
  // Cases where the result is exactly zero and its sign is not determined by the
  // rounding mode are additions of same-signed zeros.
  elseif zeroA && zeroP && signA == signP then
    result = FPZero(signA);
  // Otherwise calculate numerical result and round it.
  else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding, fpexc);
    endif
  endif
endif
if invalidop && fpexc then
FPProcessDenom

FPProcessDenorms3(typeA, type1, type2, N, fpcr);

return result;
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
boolean fpexc = TRUE; // Generate floating-point exceptions
return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr, boolean fpexc)
assert N == 32;
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,value2) = FPUnpack(op2, fpcr, fpexc);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

(done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);
if !((HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
result = FPDefaultNaN(fpcr);
if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
if !done then
infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

// Determine sign and type product will have if it does not cause an
// Invalid Operation.
signP = sign1 EOR sign2;
infp = inf1 || inf2;
zeroP = zero1 || zero2;

// Non SNaN-generated Invalid Operation cases are multiples of zero by infinity and
// additions of opposite-signed infinities.
invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
if invalidop then
result = FPDefaultNaN(fpcr);
if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
else if (infA && signA == '0') || (infP && signP == '0') then
result = FPIInfinity('0');
elsif (infA && signA == '1') || (infP && signP == '1') then
result = FPIInfinity('1');
else if zeroA && zeroP && signA == signP then
result = FPZero(signA);
else
result_value = valueA + (value1 * value2);
if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
result = FPZero(result_sign);
else
result = FPRound(result_value, fpcr, rounding, fpexc);
if !invalidop && fpexc then
FPProcessDenorm(typeA, N, fpcr);
Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================
(bool, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
    bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
    FPCRType fpcr, bool fpexc)

assert N IN {32,64};

bits(N) result;
FPType type_nan;
// When TRUE, use alternative NaN propagation rules.
bool altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
bool op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
bool op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
bool op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
if altfp then
    if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
        type_nan = FPType_SNaN;
    else
        type_nan = FPType_QNaN;
else
    type_nan = FPType_QNaN;

bool done;
if altfp && op1_nan && op2_nan && op3_nan then       // <n> register NaN selected
    done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc));
elsif altfp && op2_nan && (op1_nan || op3_nan) then     // <n> register NaN selected
    done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc));
elsif altfp && op3_nan && op1_nan then                  // <m> register NaN selected
    done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc));
elsif type1 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
elsif type3 == FPType_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
elsif type1 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
elsif type3 == FPType_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
else
    done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```
Library pseudocode for shared/functions/float/fpmulx/FPMulX

// FPMulX()
// =======

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)

  assert N IN {16,32,64};
  bits(N) result;
  boolean done;
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);

  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
  if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
      result = FPTwo(sign1 EOR sign2);
    elsif inf1 || inf2 then
      result = FPInfinity(sign1 EOR sign2);
    elsif zero1 || zero2 then
      result = FPZero(sign1 EOR sign2);
    else
      result = FPRound(value1*value2, fpcr);
  FPProcessDenorms(type1, type2, N, fpcr);

  return result;

Library pseudocode for shared/functions/float/fpneg/FPNeg

// FPNeg()
// ======

bits(N) FPNeg(bits(N) op)

  assert N IN {16,32,64};
  if !UsingAArch32() && HaveAltFP() then
    FPCRType fpcr = FPSCR[];
    if fpcr.AH == '1' then
      (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
      if fptype IN {FPType_SNaN, FPType_QNaN} then
        return op;        // When fpcr.AH=1, sign of NaN has no consequence
    return NOT(op<N-1>) : op<N-2:0>;

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

// FPOnePointFive()
// ===============

bits(N) FPOnePointFive(bit sign)

  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp = '0':Ones(E-1);
  frac = '1':Zeros(F-1);
  result = sign : exp : frac;

  return result;
Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm

// FPProcessDenorm()
// ===============
// Handles denormal input in case of single-precision or double-precision  
// when using alternative floating-point mode.

FPProcessDenorm(FPType fptype, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms

// FPProcessDenorms()
// ===============
// Handles denormal input in case of single-precision or double-precision  
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, FPType type2, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms3

// FPProcessDenorms3()
// ===============
// Handles denormal input in case of single-precision or double-precision  
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms4

// FPProcessDenorms4()
// ===============
// Handles denormal input in case of single-precision or double-precision  
// when using alternative floating-point mode.

FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
// FPProcessException()
// ====================
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRTYPE fpcr)

integer cumul;
// Determine the cumulative exception bit number
case exception of
  when FPExc_InvalidOp       cumul = 0;
  when FPExc_DivideByZero    cumul = 1;
  when FPExc_Overflow        cumul = 2;
  when FPExc_Underflow       cumul = 3;
  when FPExc_Inexact         cumul = 4;
  when FPExc_InputDenorm     cumul = 7;

enable = cumul + 8;
if fpcr<enable> == '1' then
  // Trapping of the exception enabled.
  // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
  // and if so then how exceptions and in what order that they may be
  // accumulated before calling FPTrappedException().
  bits(8) accumulated_exceptions = GetAccumulatedFPExceptions();
  accumulated_exceptions<cumul> = '1';
  if boolean IMPLEMENTATION DEFINED "Process floating-point exception" then
    if UsingAArch32() then
      AArch32.FPTrappedException(accumulated_exceptions);
    else
      is_ase = IsASEInstruction();
      AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    else
      // The exceptions generated by this instruction are accumulated by the PE and
      // FPTrappedException is called later during its execution, before the next
      // instruction is executed. This field is cleared at the start of each FP instruction.
      SetAccumulatedFPExceptions(accumulated_exceptions);
  else
    if UsingAArch32() then
      // Set the cumulative exception bit
      FPSCR<cumul> = '1';
    else
      // Set the cumulative exception bit
      FPSR<cumul> = '1';

  return;
Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr, boolean fpexc)
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;
    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;
    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPEx.INVALIDOP, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(fpcr);
    return result;
// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfmaxfmin = FALSE; // Do not use alfp mode for FMIN, FMAX and variants
    boolean fpexc       = TRUE;  // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);

// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean altfmaxfmin, boolean fpexc)
    assert N IN {16,32,64};
    bit sign2;
    boolean done;
    bits(N) result;
    boolean altfp    = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan  = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan  = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType type_nan = if any_snan then FPType_SNaN else FPType_QNaN;
    if altfmaxfmin && (op1_nan || op2_nan) then
        FPProcessException(FPExc_InvalidOp, fpcr);
        done = TRUE; sign2 = op2<N-1>;
        result = if type2 == FPType_Zero then FPZero(sign2) else op2;
    elsif altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE; result = FPProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
Library pseudocode for shared/functions/float/fpprocessnans3/FProcessNaNs3

// FProcessNaNs3()
// ================

(boolean, bits(N)) FProcessNaNs3(FPType type1, FPType type2, FPType type3,
bits(N) op1, bits(N) op2, bits(N) op3,
FPCRType fpcr)

boolean fpexc = TRUE;  // Generate floating-point exceptions
return FProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FProcessNaNs3()
// ================

// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FProcessNaNs3(FPType type1, FPType type2, FPType type3,
bits(N) op1, bits(N) op2, bits(N) op3,
FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};
bits(N) result;
boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
FPType type_nan;
if altfp then
  if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
    type_nan = FPType_SNaN;
  else
    type_nan = FPType_QNaN;
else
  done = FALSE;  result = Zeros();  // 'Don't care' result

return (done, result);
// FPRecipEstimate()
// ================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr_in)

assert N IN {16,32,64};
FPCRType fpcr = fpcr_in;
bits(N) result;
boolean overflow_to_inf;
// When using alternative floating-point behaviour, do not generate
// floating-point exceptions, flush denormal input and output to zero,
// and use RNE rounding mode.
boolean altfp = HaveAltFP() & & !UsingAArch32() & & fpcr.AH == '1';
boolean fpexc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';
if altfp then fpcr.RMode = '00';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

FPRounding rounding = FPRoundingMode(fpcr);
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elself fptype == FPTYPE_INFINITY then
result = FPInfinity(sign);
elself fptype == FPTYPE_ZERO then
result = FPZero(sign);
elself (N == 16 & & Abs(value) < 2.0^-16) ||
(N == 32 & & Abs(value) < 2.0^-128) ||
(N == 64 & & Abs(value) < 2.0^-1024)
then
// Result flushed to zero of correct sign
result = FPZero(sign);

else if (fpcr.FZ == '1' & & N != 16) || (fpcr.FZ16 == '1' & & N == 16))
& &
(N == 16 & & Abs(value) >= 2.0^14) ||
(N == 32 & & Abs(value) >= 2.0^126) ||
(N == 64 & & Abs(value) >= 2.0^1022)
then
// Result flushed to zero of correct sign
result = FPZero(sign);

else
// Flush-to-zero never generates a trapped exception.
if UsingAArch32() then
FPSCR.UFC = '1';
elself if fpexc then FPSR.UFC = '1';
else
// Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
// calculate result exponent. Scaled value has copied sign bit,
// exponent = 1022 = double-precision biased version of -1,
// fraction = original fraction
bits(52) fraction;
integer exp;
case N of
when 16
fraction = operand<9:0> : Zeros(42);
exp = UInt(operand<14:10>);
when 32
  fraction = operand<22:0> : Zeros(29);
  exp = UInt(operand<30:23>);
when 64
  fraction = operand<51:0>;
  exp = UInt(operand<62:52>);

if exp == 0 then
  if fraction<51> == '0' then
    exp = -1;
    fraction = fraction<49:0>:'00';
  else
    fraction = fraction<50:0>:'0';

integer scaled;
boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

if !increasedprecision then
  scaled = UInt('1':fraction<51:44>);
else
  scaled = UInt('1':fraction<51:41>);

integer result_exp;

case N of
  when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 29+1 = 30
  when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
  when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

// Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
// fixed-point number in range [0.5 .. 1.0].
estimate = RecipEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
  fraction = estimate<7:0> : Zeros(44);
else
  fraction = estimate<11:0> : Zeros(40);

if result_exp == 0 then
  fraction = '1' : fraction<51:1>;
elsif result_exp == -1 then
  fraction = '01' : fraction<51:2>;
  result_exp = 0;

case N of
  when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
  when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
  when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;
return result;
// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedPrecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1;                       // Round to nearest
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2;                 // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1;                       // Round to nearest
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5;  // Error cannot be exactly 0.5 so do not need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;

    return r;
/Library pseudocode for shared/functions/float/fprecpx/FPRecpX

// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRTyp e fpcr_in)

assert N IN {16,32,64};
FPCRTyp e fpcr = fpcr_in;
integer esize;
case N of
    when 16 esize =  5;
    when 32 esize =  8;
    when 64 esize = 11;

bits(N)     result;
bits(esize)  exp;
bits(esize)  max_exp;
bits(N-(esize+1)) frac = Zeros();

boolean altfp = HaveAltFP() && fpcr.AH == '1';
boolean fpexc = !altfp;                 // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11';     // Flush denormal input and output to zero
(fptype,sign,value) = FPUunpack(op, fpcr, fpexc);

case N of
    when 16 exp = op<10+esize-1:10>;
    when 32 exp = op<23+esize-1:23>;
    when 64 exp = op<52+esize-1:52>;

max_exp = Ones(esize) - 1;

if fptype == FPTyp e SNaN || fptype == FPTyp e QNaN then
    result = FPProcessNaN(fptype, op, fpcr, fpexc);
else
    if IsZero(exp) then // Zero and denormals
        result = sign:max_exp:frac;
    else // Infinities and normals
        result = sign:NOT(exp):frac;

return result;
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE;  // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, boolean fpexc)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
Library pseudocode for shared/functions/float/fpround/FPRoundBase
// FPRoundBase()
// =============

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16)
bool fpexc = TRUE;    // Generate floating-point exceptions
return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRoundBase()
// =============

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding,
boolean isbfloat16, boolean fpexc)

assert N IN {16,32,64};
assert op != 0.0;
assert rounding != FPRounding_TIEAWAY;
bits(N) result;

// Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
integer minimum_exp;
integer F;
integer E;
if N == 16 then
    minimum_exp = -14;  E = 5;  F = 10;
elsif N == 32 && isbfloat16 then
    minimum_exp = -126;  E = 8;  F = 7;
elselif N == 32 then
    minimum_exp = -126;  E = 8;  F = 23;
else  // N == 64
    minimum_exp = -1022;  E = 11;  F = 52;

// Split value into sign, unrounded mantissa and exponent.
bit sign;
real mantissa;
if op < 0.0 then
    sign = '1';  mantissa = -op;
elselse
    sign = '0';  mantissa = op;
exponent = 0;
while mantissa < 1.0 do
    mantissa = mantissa * 2.0;  exponent = exponent - 1;
while mantissa >= 2.0 do
    mantissa = mantissa / 2.0;  exponent = exponent + 1;

// When TRUE, detection of underflow occurs after rounding and the test for a
// denormalized number for single and double precision values occurs after rounding.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

// Deal with flush-to-zero before rounding if FPCR.AH != '1'.
if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
exponent < minimum_exp) then
    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
        FPSCR.UFC = '1';
elselse
        if fpexc then FPSR.UFC = '1';
        return FPZero(sign);
biased_exp_unconstrained = (exponent - minimum_exp) + 1;
int mant_unconstrained = RoundDown(mantissa * 2.0^F);
error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^F; // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
if !altfp && biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
    if fpexc then FPPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then
    case rounding of
    when FPRounding_TIEEVEN
      round_up_unconstrained = (error_unconstrained > 0.5 ||
        (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
      round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
      overflow_to_inf = TRUE;
    when FPRounding_POSINF
      round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
      round_up = (error != 0.0 && sign == '0');
      overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
      round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
      round_up = (error != 0.0 && sign == '1');
      overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD
      round_up_unconstrained = FALSE;
      round_up = FALSE;
      overflow_to_inf = FALSE;
    end case
    if round_up_unconstrained then
      int_mant_unconstrained = int_mant_unconstrained + 1;
      if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
        biased_exp_unconstrained = biased_exp_unconstrained + 1;
        int_mant_unconstrained = int_mant_unconstrained DIV 2;
    end if

    // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
    if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
        // the result of unrounded rounding is less than the minimum normalized number
        if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero
            if fpexc then
                FPZero(sign);
                FPPProcessException(FPExc_Inexact, fpcr);
            elsif error != 0.0 || fpcr.UFE == '1' then
                if fpexc then FPPProcessException(FPExc_Underflow, fpcr);
            else // altfp == FALSE
                case rounding of
                when FPRounding_TIEEVEN
                    round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                    overflow_to_inf = TRUE;
                when FPRounding_POSINF
                    round_up = (error != 0.0 && sign == '0');
                    overflow_to_inf = (sign == '0');
                when FPRounding_NEGINF
                    round_up = (error != 0.0 && sign == '1');
                    overflow_to_inf = (sign == '1');
                when FPRounding_ZERO, FPRounding_ODD
                    round_up = FALSE;
                    overflow_to_inf = FALSE;
                end case

                if round_up then
                    int_mant = int_mant + 1;
                end if
        end if
    end if

if int_mant == 2^F then  // Rounded up from denormalized to normalized
    biased_exp = 1;
if int_mant == 2^{F+1} then  // Rounded up to next exponent
    biased_exp = biased_exp + 1;
int_mant = int_mant DIV 2;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding_ODD then
    int_mant<=0> = '1';

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        if fpexc then FPProcessException(FPEx_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign : biased_exp<1:0> : int_mant<1:0> : Zeros(N-(E+F+1));
    else // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            if fpexc then FPProcessException(FPEx_InvalidOp, fpcr);
            error = 0.0; // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<1:0> : int_mant<1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPProcessException(FPEx_Inexact, fpcr);
return result;

Library pseudocode for shared/functions/float/fpround/FPRoundCV

// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr_in, FPRounding rounding)
    FPCRType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

Library pseudocode for shared/functions/float/fprounding/FPRounding

enumeration FPRounding                          {
    FPRounding_TIEEVEN, FPRounding_POSINF,
    FPRounding_NEGINF, FPRounding_ZERO,
    FPRounding_TIEAWAY, FPRounding_ODD};

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)

assert rounding != FPRounding_ODD;
assert N IN {16,32,64};

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpxc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fxpc);

bits(N) result;
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPType_Infinity then
    result = FPIInfinity(sign);
elsif fptype == FPType_Zero then
    result = FPZero(sign);
else
    // Extract integer component.
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
    if round_up then int_result = int_result + 1;

    // Convert integer value into an equivalent real value.
    real_result = Real(int_result);

    // Re-encode as a floating-point value, result is always exact.
    if real_result == 0.0 then
        result = FPZero(sign);
    else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);

    // Generate inexact exceptions.
if error != 0.0 && exact then
    FPProcessException(FPExc_Inexact, fpcr);
return result;
bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
assert rounding != FPRounding_ODD;
assert N IN {32,64};
assert intsize IN {32, 64};
integer exp;
bite(N) result;
boolean round_up;
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - (E + 1);
// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;
// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
  if N == 32 then
    exp = 126 + intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
  else
    exp = 1022+intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif fptype == FPType_Zero then
    result = FPZero(sign);
  else
    // Extract integer component.
    int_result = RoundDown(value);
    error = value - Real(int_result);
    // Determine whether supplied rounding mode requires an increment.
    case rounding of
      when FPRounding_TIEEVEN
        round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
      when FPRounding_POSINF
        round_up = error != 0.0;
      when FPRounding_NEGINF
        round_up = FALSE;
      when FPRounding_ZERO
        round_up = error != 0.0 && int_result < 0;
      when FPRounding_TIEAWAY
        round_up = error > 0.5 || (error == 0.5 && int_result >= 0);
    if round_up then int_result = int_result + 1;
    overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);
    if overflow then
      if N == 32 then
        exp = 126 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
      else
        exp = 1022 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
      // This case shouldn’t set Inexact.
      error = 0.0;
    else
      // Convert integer value into an equivalent real value.
      real_result = Real(int_result);
      // Re-encode as a floating-point value, result is always exact.
      if real_result == 0.0 then
        result = FPZero(sign);
else
    result = FPRound(real_result, fpcr, FPRounding_ZERO);

// Generate inexact exceptions.
if error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);

return result;
Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate
// FPRSqrtEstimate()
// =============

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr_in)

assert N IN {16,32,64};
FPCRType fpcr = fpcr_in;

// When using alternative floating-point behaviour, do not generate
// floating-point exceptions and flush denormal input to zero.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

bits(N) result;
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elsif fptype == FPType_Zero then
    result = FPInfinity(sign);
    if fpexc then
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr);
        if fpexc then
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif fptype == FPType_Infinity then
            result = FPZero('0');
        else
            // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
            // evenness or oddness of the exponent unchanged, and calculate result exponent.
            // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
            // biased version of -1 or -2, fraction = original fraction extended with zeros.
            integer N of
                case N of
                    when 16
                        fraction = operand<9:0> : Zeros(42);
                        exp = UInt(operand<14:10>);
                    when 32
                        fraction = operand<22:0> : Zeros(29);
                        exp = UInt(operand<30:23>);
                    when 64
                        fraction = operand<51:0>;
                        exp = UInt(operand<62:52>);
                if exp == 0 then
                    while fraction<51> == '0' do
                        fraction = fraction<50:0> : '0';
                        exp = exp - 1;
                    fraction = fraction<50:0> : '0';

                integer scaled;
                boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;
                if !increasedprecision then
                    if exp<0> == '0' then
                        scaled = UInt('1':fraction<51:44>);
                    else
                        scaled = UInt('01':fraction<51:45>);
                    if exp<0> == '0' then
                        scaled = UInt('1':fraction<51:41>);
                    else
                        scaled = UInt('01':fraction<51:42>);

                integer result_exp;
                case N of
                    when 16 result_exp = (  44 - exp) DIV 2;
                    when 32 result_exp = ( 380 - exp) DIV 2;
when 64 result_exp = (3068 - exp) DIV 2;

estimate = \texttt{RecipSqrtEstimate}(\text{scaled, increasedprecision});

// Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit and high-order
// fraction bits, and exponent calculated above.
case N of
when 16 result = '0' : result_exp<N-12:0> : estimate<7:0> : \texttt{Zeros}(2);
  when 32
    if !increasedprecision then
      result = '0' : result_exp<N-25:0> : estimate<7:0> : \texttt{Zeros}(15);
    else
      result = '0' : result_exp<N-25:0> : estimate<11:0> : \texttt{Zeros}(11);
  when 64 result = '0' : result_exp<N-54:0> : estimate<7:0> : \texttt{Zeros}(44);
return result;
// RecipSqrtEstimate()
// -------------------
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)
{
    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then  // 0.25 .. 0.5
            a = a*2+1;         // a in units of 1/512 rounded to nearest
        else             // 0.5 .. 1.0
            a = (a >> 1) << 1; // Discard bottom bit
        a = (a+1)*2;       // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2;  // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;
        if a < 2048 then  // 0.25.. 0.5
            a = a*2 + 1;     // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else             // 0.5..1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = a+1;            // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);
        real_val = Sqrt(real_val);  // This number will lie in the range of 32 to 64
        real_val = real_val * Real(2^47);  // The integer is the size of the whole DP mantissa
        int_val = RoundDown(real_val);  // Calculate rounding value
        error = real_val - Real(int_val);
        round_up = error > 0.5;        // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;
        real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
        int_val = RoundDown(real_val);  // Round that (to nearest even) to give integer
        error = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
        if round_up then int_val = int_val+1;
        r = int_val;
        assert 4096 <= r && r < 8192;
    return r;
}
Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

// FPSqrt()  
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};
    (ftype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTYPE_Zero then
        result = FPZero(sign);
    elsif fptype == FPTYPE_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);
        FPProcessDenorm(fptype, N, fpcr);
    return result;

Library pseudocode for shared/functions/float/fpsub/FPSub

// FPSub()  
// ========

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE_Infinity);
        inf2 = (type2 == FPTYPE_Infinity);
        zero1 = (type1 == FPTYPE_Zero);
        zero2 = (type2 == FPTYPE_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elseif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
            FPProcessDenorms(type1, type2, N, fpcr);
        end if
    end if
    return result;
Library pseudocode for shared/functions/float/fpthree/FPTThree

```
// FPTThree()
// =========

bits(N) FPTThree(bit sign)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = '1':Zeros(F-1);
result = sign : exp : frac;

return result;
```
Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```plaintext
// FPToFixed()
// ============
// Convert N-bit precision floating point OP to M-bit fixed point with FBITS fractional bits, controlled by UNSIGNED and Rounding.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRTyppe fpcr, FPRoundingT rounding)

assert N IN {16,32,64};
assert M IN {16,32,64};
assert fbits >= 0;
assert rounding != FPRounding_ODD;

// When alternative floating-point support is TRUE, do not generate Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

// If NaN, set cumulative flag or take exception.
if fptype == FPType_SNan || fptype == FPType_ONan then
    FPProcessException(FPExc_InvalidOp, fpcr);

// Scale by fractional bits and produce integer rounded towards minus-infinity.
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
boolean round_up; case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 && int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

if round_up then int_result = int_result + 1;

// Generate saturated result and exceptions.
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPExc_InvalidOp, fpcr);
elsif error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
return result;
```

Shared Pseudocode Functions
// FPToFixedJS()
// =============
// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRTypen fpcr, boolean Is64)
    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !((HaveAltFP() && !UsingAArch32() && fpcr.AH == '1'));

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype, sign, value) = FPUnpack(op, fpcr, fpexc_idenorm);

    Z = '1';

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);

    // Determine whether supplied rounding mode requires an increment.
    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';
    if fptype == FPType_Infinity then result = 0;

    return (result<N-1:0>, Z);

Library pseudocode for shared/functions/float/fptwo/FPTwo

// FPTwo()
// ========

bits(N) FPTwo(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
Library pseudocode for shared/functions/float/fptype/FPType

```c
// Library pseudocode for shared/functions/float/fptype/FPType

// FPType enumeration
FPType FPType_Zero,
FPType Denormal,
FPType Nonzero,
FPType Infinity,
FPType QNaN,
FPType SNaN;
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```c
// Library pseudocode for shared/functions/float/fpunpack/FPUnpack

// FPUnpack()
// =========
(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in)
FPCRType fpcr = fpcr_in;
fpcr.AHP = '0';
boolean fpexc = TRUE;   // Generate floating-point exceptions
(fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);

// FPUnpack()
// =========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in, boolean fpexc)
FPCRType fpcr = fpcr_in;
fpcr.AHP = '0';
(fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);
```
Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase
FPUnpackBase()  
=============  

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)  
return (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);  

FPUnpackBase()  
==============  
// Unpack a floating-point number into its type, sign bit and the real number  
// that it represents. The real number result has the correct sign for numbers  
// and infinities, is very large in magnitude for infinities, and is 0.0 for  
// NaNs. (These values are chosen to simplify the description of comparisons  
// and conversions.)  
//  
// The 'fpcr_in' argument supplies FPCR control bits and 'fpexc' controls the  
// generation of floating-point exceptions. Status information is updated  
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr_in, boolean fpexc)

assert N IN {16,32,64};

FPCRType fpcr = fpcr_in;

boolean altfp = HaveAltFP() && !UsingAArch32();
boolean fiz = altfp && fpcr.FIZ == '1';
boolean fz = fpcr.FZ == '1' &&!(altfp && fpcr.AH == '1');
real value;
bit sign;
FPType fptype;

if N == 16 then
  sign = fpval<15>;
  exp16 = fpval<14:10>;
  frac16 = fpval<9:0>;
  if IsZero(exp16) then
    if IsZero(frac16) || fpcr.FZ16 == '1' then
      fptype = FPType_Zero;  value = 0.0;
    else
      fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
  elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
    if IsZero(frac16) then
      fptype = FPType_Infinity;  value = 2.0^1000000;
    else
      fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
      value = 0.0;
  else
    fptype = FPType_Nonzero;
    value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);
  end
elsif N == 32 then
  sign = fpval<31>;
  exp32 = fpval<30:23>;
  frac32 = fpval<22:0>;
  if IsZero(exp32) then
    if IsZero(frac32) then
      // Produce zero if value is zero.
      fptype = FPType.Zero;  value = 0.0;
    elsif fz || fiz then  // Flush-to-zero if FIZ==1 or AH,FZ==01
      fptype = FPType_Zero;  value = 0.0;
      // Check whether to raise Input Denormal floating-point exception.
      // fpcr.FIZ==1 does not raise Input Denormal exception.
      if fz then
        // Denormalized input flushed to zero
        if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
      else
        return (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
      fi
    end
  end
  elsif fpcr.FIZ == '1' then
    // Generate floating-point exceptions
    boolean fpexc = TRUE;
  endif
fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
elsif IsOnes(exp32) then
  if IsZero(frac32) then
    fptype = FPType_Infinity;  value = 2.0^1000000;
  else
    fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
    value = 0.0;
  else
    fptype = FPType_Nonzero;
    value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
else // N == 64
  sign   = fpval<63>;
  exp64  = fpval<62:52>;
  frac64 = fpval<51:0>;
  if IsZero(exp64) then
    if IsZero(frac64) then
      // Produce zero if value is zero.
      fptype = FPType_Zero;  value = 0.0;
    elsif fz || fiz then
      // Flush-to-zero if FIZ==1 or AH,FZ==01
      fptype = FPType_Zero;  value = 0.0;
    // Check whether to raise Input Denormal floating-point exception.
    // fpcr.FIZ=1 does not raise Input Denormal exception.
    if fz then
      // Denormalized input flushed to zero
      if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
    else
      fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
    else
      if IsZero(frac64) then
        // Flush-to-zero if FIZ==1 or AH,FZ==01
        fptype = FPType_Zero;  value = 0.0;
      end else
        fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
        value = 0.0;
      else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
      if sign == '1' then value = -value;
return (fptype, sign, value);

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

// FPUnpackCV()
// ============
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.
(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr_in)
FPCRType fpcr = fpcr_in;
fpcr.FZ16 = '0';
boolean fpexc = TRUE;  // Generate floating-point exceptions
(fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);
Library pseudocode for shared/functions/float/fpzero/FPZero

```plaintext
// FPZero()
// ========
bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```

Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

```plaintext
// VFPExpandImm()
// ==============
bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>) : Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0> : Zeros(F-4);
    result = sign : exp : frac;
    return result;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```plaintext
// AddWithCarry()
// ===============
// Integer addition with carry input, returning result and NZCV flags
(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/interrupts/InterruptID

```plaintext
enumeration InterruptID {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
};
```
Library pseudocode for shared/functions/interrupts/SetInterruptRequestLevel

// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

// AArch64.BranchAddr()
// ===============
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
assert !UsingAArch32();
msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
if msbit == 63 then
  return vaddress;
elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
  return SignExtend(vaddress<msbit>);
else
  return ZeroExtend(vaddress<msbit>);

Library pseudocode for shared/functions/memory/AccType

enumeration AccType {
  AccType_NORMAL, // Normal loads and stores
  AccType_STREAM, // Streaming loads and stores
  AccType_VEC, // Vector loads and stores
  AccType_VECSTREAM, // Streaming vector loads and stores
  AccType_SVE, // Scalable vector loads and stores
  AccType_STREAMSVE, // Streaming scalable vector loads and stores
  AccType_UNPRIVSTREAM, // Streaming unprivileged loads and stores
  AccType_A32LSMD, // Load and store multiple
  AccType_ATOMIC, // Atomic loads and stores
  AccType_ATOMICR, // Atomic reads and stores
  AccType_ATOMICRW, // Atomic reads and store-release
  AccType_ATOMICRWATOMIC, // Atomic reads and store-release with atomic access
  AccType_ATOMICRWATOMICR, // Atomic reads and store-release with atomic access
  AccType_ATOMICR64, // Atomic 64-byte loads and stores
  AccType_ATOMICRLOD, // Load-LOAcquire and Store-LORelease
  AccType_ATOMICRNU, // Load and store unprivileged
  AccType_ATOMICRIF, // Instruction fetch
  AccType_ATOMICRTTW, // Translation table walk
  AccType_ATOMICRNONFAULT, // Non-faulting loads
  AccType_ATOMICRNV2REGISTER, // MRS/MSR instruction used at EL1 and which is
  // converted to a memory access that uses the
  // EL2 translation regime
  AccType_ATOMICRDC, // Data cache maintenance
  AccType_ATOMICRNC, // Non-contiguous FF load, not first element
  AccType_ATOMICRNC2MSR, // Consecutive FF read, not first element
                         // converted to a memory access that uses the
                         // EL2 translation regime
  AccType_ATOMICRDCZVA, // Address translation with PAN permission checks
  AccType_ATOMICRAT, // Address translation with PAN permission checks
};

Library pseudocode for shared/functions/memory/AccessDescriptor

type AccessDescriptor is (
  MPAMinfo mpam,
  AccType accType)

Shared Pseudocode Functions
Library pseudocode for shared/functions/memory/AddrTop

// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
assert HaveEL(el);
regime = S1TranslationRegime(el);
if ELUsingAArch32(regime) then // AArch32 translation regime.
    return 31;
extif EffectiveTBI(address, IsInstr, el) == '1' then
    return 55;
extif EffectiveTBI(address, IsInstr, el) == '0' then
    return 63;

Library pseudocode for shared/functions/memory/Allocation

constant bits(2) MemHint_No = '00'; // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01'; // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10'; // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11'; // Read-Allocate, Write-Allocate

Library pseudocode for shared/functions/memory/BigEndian

// BigEndian()
// ===========

boolean BigEndian(AccType accype)
    boolean bigend;
if HaveNV2Ext() && accype == AccType_NV2REGISTER then
    return SCTLR_EL2.EE == '1';
if UsingAArch32() then
    bigend = (PSTATE.E != '0');
extif PSTATE.EL == EL0 then
    bigend = (SCTLR[].E0E != '0');
extelse
    bigend = (SCTLR[].EE != '0');
return bigend;

Library pseudocode for shared/functions/memory/BigEndianReverse

// BigEndianReverse()
// ===============

bits(width) BigEndianReverse (bits(width) value)
assert width IN {8, 16, 32, 64, 128};
integer half = width DIV 2;
if width == 8 then return value;
if width == 8 then return value;
return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);

Library pseudocode for shared/functions/memory/Cacheability

constant bits(2) MemAttr_NC = '00'; // Non-cacheable
constant bits(2) MemAttr_WT = '10'; // Write-through
constant bits(2) MemAttr_WB = '11'; // Write-back
// CreateAccessDescriptor()
// -----------------------

AccessDescriptor CreateAccessDescriptor(AccType acctype)
{
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype);
    return accdesc;
}

// DataMemoryBarrier()
// -------------------

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

// DataSynchronizationBarrier()
// ----------------------------

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);

// DeviceType enumeration
// -----------------------

enumeration DeviceType {
    DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE;
}

// EffectiveTBI()
// ---------------

// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
{
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(! ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBID;
    
    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
}
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;
        return tcma;

Library pseudocode for shared/functions/memory/Fault


Library pseudocode for shared/functions/memory/FaultRecord

type FaultRecord is (Fault statuscode, // Fault Status
    AccType acctype, // Type of access that faulted
    FullAddress ipaddress, // Intermediate physical address
    boolean s2fs1walk, // Is on a Stage 1 translation table walk
    boolean write, // TRUE for a write, FALSE for a read
    integer level, // For translation, access flag and permission faults
    bit extflag, // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean secondstage, // Is a Stage 2 abort
    bits(4) domain, // Domain number, AArch32 only
    bits(2) errortype, // [Armv8.2 RAS] AArch32 AET or AArch64 SET
    bits(4) debugmoe) // Debug method of entry, from AArch32 only
Library pseudocode for shared/functions/memory/FullAddress

type FullAddress is (PASpace paspace, bits(52) address)

Library pseudocode for shared/functions/memory/Hint_Prefetch

// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);

Library pseudocode for shared/functions/memory/MBReqDomain

enumeration MBReqDomain {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                        MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

Library pseudocode for shared/functions/memory/MBReqTypes

enumeration MBReqTypes {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};

Library pseudocode for shared/functions/memory/MPAM

type PARTIDtype = bits(16);
type PMGtype = bits(8);
type PARTIDspaceType = bit;
constant PARTIDspaceType PIdSpace_Secure = '0';
constant PARTIDspaceType PIdSpace_NonSecure = '1';

type MPAMinfo is (PARTIDspaceType mpam_ns,
                  PARTIDtype partid,
                  PMGtype pmg)

Library pseudocode for shared/functions/memory/MemAttrHints

type MemAttrHints is (bits(2) attrs, // See MemAttr_*, Cacheability attributes
                      bits(2) hints, // See MemHint_*, Allocation hints
                      boolean transient)

Library pseudocode for shared/functions/memory/MemType

enumeration MemType {MemType_Normal, MemType_Device};
Library pseudocode for shared/functions/memory/MemoryAttributes

type MemoryAttributes is (
    MemType memtype,
    DeviceType device, // For Device memory types
    MemAttrHints inner, // Inner hints and attributes
    MemAttrHints outer, // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    boolean tagged, // Tagged access
    bit xs // XS attribute
)

Library pseudocode for shared/functions/memory/PASpace

enumeration PASpace {
    PAS_NonSecure,
    PAS_Secure,
};

Library pseudocode for shared/functions/memory/Permissions

type Permissions is (
    bits(2) ap_table,   // Stage 1 hierarchical access permissions
    bit xn_table,       // Stage 1 hierarchical execute-never for single EL regimes
    bit pxn_table,      // Stage 1 hierarchical privileged execute-never
    bit uxn_table,      // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,         // Stage 1 access permissions
    bit xn,             // Stage 1 execute-never for single EL regimes
    bit uxn,            // Stage 1 unprivileged execute-never
    bit pxn,            // Stage 1 privileged execute-never
    bits(2) s2ap,       // Stage 2 access permissions
    bit s2xn,           // Stage 2 extended execute-never
    bit s2xn,           // Stage 2 execute-never
)

Library pseudocode for shared/functions/memory/PhysMemRead

// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an external abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.
(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
     AccessDescriptor accdesc);

Library pseudocode for shared/functions/memory/PhysMemRetStatus

type PhysMemRetStatus is (Fault statuscode, // Fault Status
    bit extflag, // IMPLEMENTATION DEFINED
    bits(2) errortype, // syndrome for External aborts
    bits(64) store64bstatus, // status of 64B store
    AccType acctype) // Type of access that faulted

Library pseudocode for shared/functions/memory/PhysMemWrite

// Writes the value to memory, and returns the status of the write.
// If there is an external abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
                          bits(8*size) value);
Library pseudocode for shared/functions/memory/PrefetchHint

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};

Library pseudocode for shared/functions/memory/Shareability

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

SpeculativeStoreBypassBarrierToPA();

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

SpeculativeStoreBypassBarrierToVA();

Library pseudocode for shared/functions/memory/Tag

constant integer LOG2_TAG_GRANULE = 4;
constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;

Library pseudocode for shared/functions/mpam/DefaultMPAMinfo

// DefaultMPAMinfo()
// ================
// Returns default MPAM info. The partidspace argument sets
// the PARTID space of the default MPAM information returned.

MPAMinfo DefaultMPAMinfo(PARTIDspaceType partidspace) {
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_ns = partidspace;
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
}

Library pseudocode for shared/functions/mpam/DefaultPARTID

constant PARTIDtype DefaultPARTID = 0<15:0>;

Library pseudocode for shared/functions/mpam/DefaultPMG

constant PMGtype DefaultPMG = 0<7:0>;}
// GenMPAMcurEL()
// ==============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccType acctype)
{
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = if IsSecure() then SS_Secure else SS_NonSecure;
    boolean InD = FALSE;
    PARTIDspaceType pspace = PARTIDspaceFromSS(security);
    if pspace == PiDSpace_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
        validEL = TRUE;
    case acctype of
    when AccType_IFETCH, AccType_IC
        InD = TRUE;
    otherwise
        // Other access types are DATA accesses
        InD = FALSE;
    if !validEL then
        return DefaultMPAMinfo(pspace);
    if HaveEMPAMExt() && security == SS_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            pspace = PiDSpace_NonSecure;
        if MPAM3_EL3.SDEFLT == '1' then
            return DefaultMPAMinfo(pspace);
    if !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    else
        return genMPAM(mpamEL, InD, pspace);
Library pseudocode for shared/functions/mpam/MAP_vPARTID

// MAP_vPARTID()
// ============
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
// should not ever be called if EL2 is not implemented
// or is implemented but not enabled in the current
// security state.
PARTIDtype ret;
boolean err;
integer virt = UInt(vpartid);
integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

// vpartid_max is largest vpartid supported
integer vpartid_max = (vpmrmax << 2) + 3;

// One of many ways to reduce vpartid to value less than vpartid_max.
if UInt(vpartid) > vpartid_max then
    virt = virt MOD (vpartid_max+1);

// Check for valid mapping entry.
if MPAMVPMV_EL2<virt> == '1' then
    // vpartid has a valid mapping so access the map.
    ret = mapvpmw(virt);
    err = FALSE;

// Is the default virtual PARTID valid?
elsiif MPAMVPMV_EL2<v0> == '1' then
    // Yes, so use default mapping for vpartid == 0.
    ret = MPAMVPM0_EL2<v0 +: 16>;
    err = FALSE;

// Neither is valid so use default physical PARTID.
else
    ret = DefaultPARTID;
    err = TRUE;

// Check that the physical PARTID is in-range.
// This physical PARTID came from a virtual mapping entry.
integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
if UInt(ret) > partid_max then
    // Out of range, so return default physical PARTID
    ret = DefaultPARTID;
    err = TRUE;
return (ret, err);

Library pseudocode for shared/functions/mpam/MPAMisEnabled

// MPAMisEnabled()
// =============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
el = HighestEL();
case el of
    when EL3 return MPAM3_EL3.MPAMEN == '1';
    when EL2 return MPAM2_EL2.MPAMEN == '1';
    when EL1 return MPAM1_EL1.MPAMEN == '1';
Library pseudocode for shared/functions/mpam/MPAMisVirtual

// MPAMisVirtual()
//================
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
        ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
            (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
        (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));

Library pseudocode for shared/functions/mpam/PARTIDspaceFromSS

// PARTIDspaceFromSS()
//===============
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
case security of
    when SS_NonSecure
        return PIdSpace_NonSecure;
    when SS_Secure
        return PIdSpace_Secure;
    otherwise
        Unreachable();

Library pseudocode for shared/functions/mpam/genMPAM

// genMPAM()
//===========
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID I and PMG I fields
// of MPAMEl_ELx register and otherwise using PARTID D and PMG D fields.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, PARTIDspaceType pspace)
    MPAMinfo returninfo;
    PARTIDtype partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use El1 the virtual machine's PARTIDs.
    boolean gstplk = (el == EL0 && EL2Enabled() &&
        MPAMHCR_EL2.GSTAPP_PLK == '1' &&
        HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returninfo.mpam_ns = pspace;
    returninfo.partid = partidel;
    returninfo.pmg = groupel;
    return returninfo;
Library pseudocode for shared/functions/mpam/genMPAMel

```c
// genMPAMel()
// ===========
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMel(bits(2) el, boolean InD)
    SecurityState security = SecurityStateAtEL(el);
    PARTIDspaceType space = PARTIDspaceFromSS(security);
    boolean use_default = !((HaveMPAMExt() && MPAMisEnabled()));
    if HaveEMPAMExt() && security == SS_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            space = PIdSpace_NonSecure;
        if MPAM3_EL3.SDEFLT == '1' then
            use_default = TRUE;
        if !use_default then
            return genMPAM(el, InD, space);
        else
            return DefaultMPAMinfo(space);
    else
        return DefaultMPAMinfo(space);
```

Library pseudocode for shared/functions/mpam/genPARTID

```c
// genPARTID()
// ===========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

Library pseudocode for shared/functions/mpam/genPMG

```c
// genPMG()
// =========
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```
Library pseudocode for shared/functions/mpam/getMPAM_PARTID

```c
// getMPAM_PARTID()
// ================
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD)
{
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InD then
    {
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    }
    else
    {
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_D;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
            when '01' partid = MPAM1_EL1.PARTID_D;
            when '00' partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    }

    return partid;
}
```

Library pseudocode for shared/functions/mpam/getMPAM_PMG

```c
// getMPAM_PMG()
// =============
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD)
{
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
    {
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    }
    else
    {
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_D;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
            when '01' pmg = MPAM1_EL1.PMG_D;
            when '00' pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
    }

    return pmg;
}
```
Library pseudocode for shared/functions/mpam/mapvpmw

// mapvpmw()
// =========
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros;
    end;

    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;

Library pseudocode for shared/functions/predictionrestrict/ASID

// ASID[]
// ======
// Effective ASID.

bits(16) ASID[]
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H, TGE> == '11' then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;
        end;
    elsif !ELUsingAArch32(EL1) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;
        end;
    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtend(TTBR0.ASID, 16);
            end;
        end;
    end;

Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt

type ExecutionCntxt is ( 
    boolean is_vmid_valid, // is vmid valid for current context
    boolean all_vmid,      // should the operation be applied for all vmids
    bits(16) vmid,         // if all vmid = FALSE, vmid to which operation is applied
    boolean is_asid_valid, // is asid valid for current context
    boolean all_asid,      // should the operation be applied for all asids
    bits(16) asid,         // if all asid = FALSE, ASID to which operation is applied
    bits(2) target_el,     // target EL at which operation is performed
    SecurityState security,
    RestrictType restriction // type of restriction operation
)
Library pseudocode for shared/functions/predictionrestrict/RESTRICT_PREDICTIONS

```c
// RESTRICT_PREDICTIONS()
// ================
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/predictionrestrict/RestrictType

```c
enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch
};
```

Library pseudocode for shared/functions/predictionrestrict/TargetSecurityState

```c
// TargetSecurityState()
// ================
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS)
    curr_ss = SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;
    return;
```

Library pseudocode for shared/functions/registers/BranchTo

```c
// BranchTo()
// ==============
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>);
        _PC = target_vaddress;
    return;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```c
// BranchToAddr()
// ==============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```
Library pseudocode for shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};    // Other
```

Library pseudocode for shared/functions/registers/Hint_Branch

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

Library pseudocode for shared/functions/registers/NextInstrAddr

```
// Return address of the sequentially next instruction.
bits(N) NextInstrAddr();
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 & & UsingAArch32());
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
array bits(64) _R[0..30];
```
Library pseudocode for shared/functions/sysregisters/SPSR

// SPSR[] - non-assignment form
// -----------------------------------

bits(N) SPSR[]
    bits(N) result;
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ     result = SPSR_fiq<N-1:0>;
            when M32_IRQ     result = SPSR_irq<N-1:0>;
            when M32_Svc     result = SPSR_svc<N-1:0>;
            when M32_Monitor result = SPSR_mon<N-1:0>;
            when M32_Abort   result = SPSR_abt<N-1:0>;
            when M32_Hyp     result = SPSR_hyp<N-1:0>;
            when M32_Undef   result = SPSR_und<N-1:0>;
            otherwise       Unreachable();
        else
            assert N == 64;
            case PSTATE.EL of
                when EL1        result = SPSR_EL1<N-1:0>;
                when EL2        result = SPSR_EL2<N-1:0>;
                when EL3        result = SPSR_EL3<N-1:0>;
                otherwise      Unreachable();
            return result;
    else
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ     SPSR_fiq = ZeroExtend(value);
            when M32_IRQ     SPSR_irq = ZeroExtend(value);
            when M32_Svc     SPSR_svc = ZeroExtend(value);
            when M32_Monitor SPSR_mon = ZeroExtend(value);
            when M32_Abort   SPSR_abt = ZeroExtend(value);
            when M32_Hyp     SPSR_hyp = ZeroExtend(value);
            when M32_Undef   SPSR_und = ZeroExtend(value);
            otherwise       Unreachable();
        else
            assert N == 64;
            case PSTATE.EL of
                when EL1        SPSR_EL1 = ZeroExtend(value);
                when EL2        SPSR_EL2 = ZeroExtend(value);
                when EL3        SPSR_EL3 = ZeroExtend(value);
                otherwise      Unreachable();
            return;
    return;

// SPSR[] - assignment form
// -----------------------------------

SPSR[] = bits(N) value
if UsingAArch32() then
    assert N == 32;
    case PSTATE.M of
        when M32_FIQ     SPSR_fiq = ZeroExtend(value);
        when M32_IRQ     SPSR_irq = ZeroExtend(value);
        when M32_Svc     SPSR_svc = ZeroExtend(value);
        when M32_Monitor SPSR_mon = ZeroExtend(value);
        when M32_Abort   SPSR_abt = ZeroExtend(value);
        when M32_Hyp     SPSR_hyp = ZeroExtend(value);
        when M32_Undef   SPSR_und = ZeroExtend(value);
        otherwise       Unreachable();
    else
        assert N == 64;
        case PSTATE.EL of
            when EL1        SPSR_EL1 = ZeroExtend(value);
            when EL2        SPSR_EL2 = ZeroExtend(value);
            when EL3        SPSR_EL3 = ZeroExtend(value);
            otherwise      Unreachable();
        return;
    return;

Library pseudocode for shared/functions/system/ArchVersion

equation ArchVersion {
    ARMv8p0 ,
    ARMv8p1 ,
    ARMv8p2 ,
    ARMv8p3 ,
    ARMv8p4 ,
    ARMv8p5 ,
    ARMv8p6 ,
    ARMv8p7 ,
    ARMv8p8
};
Library pseudocode for shared/functions/system/BranchTargetCheck

// BranchTargetCheck()
// ================
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.
BranchTargetCheck()
assert HaveBTIExt() && !UsingAArch32();

// The branch target check considers two state variables:
// * InGuardedPage, which is evaluated during instruction fetch.
// * BTypeCompatible, which is evaluated during instruction decode.
if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
    bits(64) pc = ThisInstrAddr();
    AArch64.BranchTargetException(pc<51:0>); 

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr = AArch64.ExecutingBTIInstr(); 

    // PSTATE.BTYPE defaults to 00 for instructions that do not explictly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';

Library pseudocode for shared/functions/system/ClearEventRegister

// ClearEventRegister()
// ===============
// Clear the Event Register of this PE.
ClearEventRegister()
    EventRegister = '0';
    return;

Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

// ClearPendingPhysicalSError();

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

// ClearPendingVirtualSError();
Library pseudocode for shared/functions/system/ConditionHolds

// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
  // Evaluate base condition.
  boolean result;
  case cond<3:1> of
    when '000' result = (PSTATE.Z == '1');                          // EQ or NE
    when '001' result = (PSTATE.C == '1');                          // CS or CC
    when '010' result = (PSTATE.N == '1');                          // MI or PL
    when '011' result = (PSTATE.V == '1');                          // VS or VC
    when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
    when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
    when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
    when '111' result = TRUE;                                       // AL
  // Condition flag values in the set '111x' indicate always true
  // Otherwise, invert condition if necessary.
  if cond<0> == '1' && cond != '1111' then
    result = !result;
  return result;

Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

ConsumptionOfSpeculativeDataBarrier();

Library pseudocode for shared/functions/system/CurrentInstrSet

// CurrentInstrSet()
// ================

InstrSet CurrentInstrSet()
//------------------

InstrSet result;
if UsingAArch32() then
  result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
  // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
else
  result = InstrSet_A64;
return result;

Library pseudocode for shared/functions/system/CurrentPL

// CurrentPL()
// =========

PrivilegeLevel CurrentPL()
  return PLoFEL(PSTATE.EL);

Library pseudocode for shared/functions/system/CurrentSecurityState

// CurrentSecurityState()
// ========================
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
  return SecurityStateAtEL(PSTATE.EL);

Library pseudocode for shared/functions/system/DSBAlias

enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
**Library pseudocode for shared/functions/system/EL0**

```plaintext
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

**Library pseudocode for shared/functions/system/EL2Enabled**

```plaintext
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
return HaveEL(EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecureEL2Enabled());
```

**Library pseudocode for shared/functions/system/ELFromM32**

```plaintext
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
// Convert an AArch32 mode encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
// and the current value of SCR.NS/SCR_EL3.NS.
// 'EL' is the Exception level decoded from 'mode'.

bits(2) el;
boolean valid = !BadMode(mode); // Check for modes that are not valid for this implementation
case mode of
  when M32_Monitor
    el = EL3;
  when M32_Hyp
    el = EL2;
    valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
  when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
    // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
    // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
    // AArch64, then these modes are EL1 modes.
    el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
  when M32_User
    el = EL0;
  otherwise
    valid = FALSE; // Passed an illegal mode value
if !valid then el = bits(2) UNKNOWN;
return (valid, el);
```
Library pseudocode for shared/functions/system/ELFromSPSR

// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'   is the Exception level decoded from 'spsr'.

(boolean, bits(2)) ELFromSPSR(bits(N) spsr)
bits(2) el;
boolean valid;
if spsr<4> == '0' then                      // AArch64 state
  el = spsr<3:2>;
else if !HaveAArch64() then                  // No AArch64 support
  valid = FALSE;
else if !HaveEL(el) then                  // Exception level not implemented
  valid = FALSE;
else if spsr<1> == '1' then               // M[1] must be 0
  valid = FALSE;
else if el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
  valid = FALSE;
else if el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
  valid = FALSE;                      // Unless Secure EL2 is enabled, EL2 only valid in Non-secure
else
  valid = TRUE;
else
  valid = FALSE;
endif
if !valid then el = bits(2) UNKNOWN;
return (valid,el);

Library pseudocode for shared/functions/system/ELIsInHost

// ELIsInHost()
// =============

boolean ELIsInHost(bits(2) el)
if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
  return FALSE;
else
  case el of
    when EL3
      return FALSE;
    when EL2
      return EL2Enabled() && HCR_EL2.E2H == '1';
    when EL1
      return FALSE;
    when EL0
      return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
    otherwise
      Unreachable();
  endcase
endif

Library pseudocode for shared/functions/system/ELStateUsingAArch32

// ELStateUsingAArch32()
// ================

boolean ELStateUsingAArch32K(EL3, boolean secure)
// See ELStateUsingAArch32K() for description. Must only be called in circumstances where
// result is valid (typically, that means 'el IN {EL1,EL2,EL3}')
(known, aarch32) = ELStateUsingAArch32K(EL3, secure);
assert known;
return aarch32;
Library pseudocode for shared/functions/system/ELStateUsingAArch32K

// ELStateUsingAArch32K()
// =============

(boolean, boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)

// Returns (known, aarch32):
//   'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
//   using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
//   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
if !HaveAArch32EL(el) then
  return (TRUE, FALSE);                      // Exception level is using AArch64
elsif secure && el == EL2
  return (TRUE, FALSE);                      // Secure EL2 is using AArch64
elsif !HaveAArch64() then
  return (TRUE, TRUE);                       // Highest Exception level, and therefore all levels are using AArch64

// Remainder of function deals with the interprocessing cases when highest Exception level is using AArch64
boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;
aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '0');
aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && HCR_EL2.RW == '0' && !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt())));

if el == EL0 && !aarch32_at_el1 then
  if PSTATE.EL == EL0 then
    aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
  else
    known = FALSE;                     // EL0 state is UNKNOWN
else
  aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1, EL0});

if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);

Library pseudocode for shared/functions/system/ELUsingAArch32

// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
return ELStateUsingAArch32K(el, IsSecureBelowEL3());

Library pseudocode for shared/functions/system/ELUsingAArch32K

// ELUsingAArch32K()
// ===============

(boolean, boolean) ELUsingAArch32K(bits(2) el)
return ELStateUsingAArch32K(el, IsSecureBelowEL3());

Library pseudocode for shared/functions/system/EffectiveTGE

// EffectiveTGE()
// ===============

// Returns effective TGE value
bit EffectiveTGE()
  if EL2Enabled() then
    return if ELUsingAArch32K(EL2) then HCR.TGE else HCR_EL2.TGE;
  else
    return '0';                           // Effective value of TGE is zero
// Terminate processing of the current instruction.
EndOfInstruction();

// PE enters a low-power state.
EnterLowPowerState();

bits(1) EventRegister;

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};

// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.
(boolean, boolean) FIQPending();

bits(8) GetAccumulatedFPExceptions();
Library pseudocode for shared/functions/system/GetPSRFromPSTATE

// GetPSRFromPSTATE()
// =================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
    if UsingAArch32() & (targetELState IN {AArch32_NonDebugState, DebugState}) then
        assert N == 32;
    else
        assert N == 64;
    bits(N) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20> = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr<27> = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
        if HaveDITExt() then
            if targetELState == AArch32_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else // AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
            spsr<19:16> = PSTATE.GE;
            spsr<15:10> = PSTATE.IT<7:2>;
            spsr<9> = PSTATE.E;
            assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
            spsr<4:0> = PSTATE.M;
        else // AArch64 state
            if HaveMTEExt() then spsr<25> = PSTATE.TCO;
            if HaveDITExt() then spsr<24> = PSTATE.DIT;
            if HaveUAOExt() then spsr<23> = PSTATE.UAO;
            spsr<21> = PSTATE.SS;
            if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
            if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
            if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
            spsr<9:6> = PSTATE.<D,A,I,F>;
            spsr<4> = PSTATE.nRW;
            spsr<3:2> = PSTATE.EL;
            spsr<0> = PSTATE.SP;
    return spsr;

Library pseudocode for shared/functions/system/HasArchVersion

// HasArchVersion()
// ================
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAArch32

// HaveAArch32()
// ===========
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return boolean IMPLEMENTATION_DEFINED "AArch32 state is supported at at least EL0";
Library pseudocode for shared/functions/system/HaveAArch32EL

```c
// HaveAArch32EL()
// ===============
// boolean HaveAArch32EL(bits(2) el)
// // Return TRUE if Exception level 'el' supports AArch32 in this implementation
// if !HaveEL(el) then
// return FALSE;                    // The Exception level is not implemented
// elseif !HaveAArch32() then
// return FALSE;                    // No Exception level can use AArch32
// elseif !HaveAArch64() then
// return TRUE;                     // All Exception levels are using AArch32
// elseif el == HighestEL() then
// return FALSE;                    // The highest Exception level is using AArch64
// elseif el == EL0 then
// return TRUE;                     // EL0 must support using AArch32 if any AArch32
// return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch64

```c
// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64";
```

Library pseudocode for shared/functions/system/HaveEL

```c
// HaveEL()
// ========
// boolean HaveEL(bits(2) el)
// if el IN {EL1, EL0} then
// return TRUE;                             // EL1 and EL0 must exist
// return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```c
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)
case el of
    when EL3
        assert secure;
        return HaveEL(EL3);
    when EL2
        if secure then
            return HaveEL(EL2) && HaveSecureEL2Ext();
        else
            return HaveEL(EL2);
    otherwise
        return (HaveEL(EL3) ||
            (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```
Library pseudocode for shared/functions/system/HaveFP16Ext

// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HighestEL

// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;

Library pseudocode for shared/functions/system/Hint_DGH

// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
// Hint_WFE()
// =========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WFxType wfxtype)
  if IsEventRegisterSet() then
    ClearEventRegister();
  elsif HaveFeatWFxT() && LocalTimeoutEvent(localtimeout) then
    // No further operation if the local timeout has expired.
    EndOfInstruction();
  else
    bits(2) target_el;
    trap = FALSE;
    if PSTATE.EL == EL0 then
      // Check for traps described by the OS which may be EL1 or EL2.
      if HaveTWEDExt() then
        sctlr = SCTLR[];
        trap = sctlr.nTWE == '0';
        target_el = EL1;
      else
        AArch64.CheckForWFxTrap(EL1, wfxtype);
      fi
      if !trap && PSTATE.EL IN (EL0, EL1) && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        if HaveTWEDExt() then
          trap = HCR_EL2.TWE == '1';
          target_el = EL2;
        else
          AArch64.CheckForWFxTrap(EL2, wfxtype);
        fi
      fi
      if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        if HaveTWEDExt() then
          trap = SCR_EL3.TWE == '1';
          target_el = EL3;
        else
          AArch64.CheckForWFxTrap(EL3, wfxtype);
        fi
      fi
      if trap && PSTATE.EL != EL3 then
        (delay_enabled, delay) = WFETrapDelay(target_el);
        // (If trap delay is enabled, Delay amount)
        if !WaitForEventUntilDelay(delay_enabled, delay) then
          // Event did not arrive before delay expired
          AArch64.WFxTrap(wfxtype, target_el);
        fi
      fi
      else
        WaitForEvent(localtimeout);
    fi
  fi
Library pseudocode for shared/functions/system/Hint_WFI

// Hint_WFI()
// =========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WFxType wfxtype)
    if InterruptPending() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch64.CheckForWFxTrap(EL1, wfxtype);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, wfxtype);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, wfxtype);
        WaitForInterrupt(localtimeout);

Library pseudocode for shared/functions/system/Hint_Yield

// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();

Library pseudocode for shared/functions/system/IRQPending

// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.
(boolean, boolean) IRQPending();
Library pseudocode for shared/functions/system/IllegalExceptionReturn

// IllegalExceptionReturn()
// ================

boolean IllegalExceptionReturn(bits(N) spsr)
{
    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
        return FALSE;
}

Library pseudocode for shared/functions/system/InstrSet

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

InstructionSynchronizationBarrier();
// InterruptPending()
// ==================
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
{
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
        IsPhysicalSErrorPending());

    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1');
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;
    
    return pending_physical_interrupt || pending_virtual_interrupt;
}

// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();

// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CIGDVAC, or DC CIGVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.

boolean IsCMOWControlledInstruction();

// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
{
    return EventRegister == '1';
}

// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
{
    return HighestEL() == el;
}

// IsInHost()
// =========

boolean IsInHost()
{
    return ELIsInHost(PSTATE.EL);
}

// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
Library pseudocode for shared/functions/system/IsSErrorEdgeTriggered

// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;
        else
            if ELUsingAArch32(target_el) then
                if syndrome<11:10> != '00' then
                    // AArch32 and not Uncontainable.
                    return TRUE;
                else
                    if syndrome<24> == '0' && syndrome<5:0> != '000000' then
                        // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor Uncategorized.
                        return TRUE;
                    else
                        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
            else
                return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
    else
        return FALSE;

Library pseudocode for shared/functions/system/IsSecure

// IsSecure()
// ===========
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();

Library pseudocode for shared/functions/system/IsSecureBelowEL3

// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL EL3) then
        return SCR.Gen().NS == '0';
    elsif HaveEL EL2) && (HaveSecureEL2Ext() || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
// IsSecureEL2Enabled()
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.
boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
            return TRUE;
        else
            return FALSE;
    else
        return IsSecure();
    else
        return FALSE;

// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();

// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();

// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.
boolean LocalTimeoutEvent(integer localtimeout);

// NonSecureOnlyImplementation()
// Returns TRUE if the security state is always Non-secure for this implementation.
boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";

// PLOfEL()
// PrivilegeLevel PLOfEL(bits(2) el)
case el of
    when EL3 return if !HaveAArch64() then PL1 else PL3;
    when EL2 return PL2;
    when EL1 return PL1;
    when EL0 return PL0;
Library pseudocode for shared/functions/system/PSTATE

PSTATE ProcState;

Library pseudocode for shared/functions/system/PhysicalCountInt

// PhysicalCountInt()
// ===========
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;

Library pseudocode for shared/functions/system/PrivilegeLevel

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};

Library pseudocode for shared/functions/system/ProcState

type ProcState is (
    bits (1) N,   // Negative condition flag
    bits (1) Z,   // Zero condition flag
    bits (1) C,   // Carry condition flag
    bits (1) V,   // Overflow condition flag
    bits (1) D,   // Debug mask bit                      [AArch64 only]
    bits (1) A,   // SError interrupt mask bit
    bits (1) I,   // IRQ mask bit
    bits (1) F,   // FIQ mask bit
    bits (1) PAN, // Privileged Access Never Bit       [v8.1]
    bits (1) UAO, // User Access Override            [v8.2]
    bits (1) DIT, // Data Independent Timing         [v8.4]
    bits (1) TCO, // Tag Check Override              [v8.5, AArch64 only]
    bits (2) BTYPE, // Branch Type                   [v8.5]
    bits (1) ALLINT, // Interrupt mask bit
    bits (1) SS,  // Software step bit
    bits (1) IL,  // Illegal Execution state bit
    bits (2) EL,  // Exception level
    bits (1) nRW, // not Register Width: 0=64, 1=32
    bits (1) SP,  // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,   // Cumulative saturation flag       [AArch32 only]
    bits (4) GE,  // Greater than or Equal flags     [AArch32 only]
    bits (1) SSBS, // Speculative Store Bypass Safe
    bits (8) IT,  // If-then bits, RES0 in CPSR       [AArch32 only]
    bits (1) J,   // J bit, RES0                     [AAArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,   // T32 bit, RES0 in CPSR           [AArch32 only]
    bits (1) E,   // Endianness bit                 [AArch32 only]
    bits (5) M    // Mode field                     [AArch32 only]
)

Library pseudocode for shared/functions/system/RestoredITBits

```c
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;
    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unexpected_ILZEROIT) then return '00000000';
        else return it;
    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && !IsZero(it<3:0>) then
        return '00000000';
    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTRL.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

Library pseudocode for shared/functions/system/SCRType

```c
type SCRTYPE;
```

Library pseudocode for shared/functions/system/SCR_GEN

```c
// SCR_GEN[]
// =========

SCRTYPE SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SR);
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SecureOnlyImplementation

```c
// SecureOnlyImplementation()
// ================
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

Library pseudocode for shared/functions/system/SecurityState

```c
enumeration SecurityState {
    SS_NonSecure,
    SS_Secure
};
```
Library pseudocode for shared/functions/system/SecurityStateAtEL

// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
if !HaveEL(EL3) then
  if SecureOnlyImplementation() then
    return SS_Secure;
  else
    return SS_NonSecure;
elsif EL == EL3 then
  return SS_Secure;
else
  // For EL2 call only when EL2 is enabled in current security state
  assert(EL != EL2 || EL2Enabled());
  if !ELUsingAArch32(EL3) then
    if SCR_EL3.NS == '1' then
      return SS_NonSecure else SS_Secure;
  else
    if SCR.NS == '1' then
      return SS_NonSecure else SS_Secure;

Library pseudocode for shared/functions/system/SendEvent

// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.
SendEvent();

Library pseudocode for shared/functions/system/SendEventLocal

// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
  EventRegister = '1';
  return;

Library pseudocode for shared/functions/system/SetAccumulatedFPExceptions

// Stores FP Exceptions accumulated by the PE.
SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```c
// SetPSTATEFromPSR() 
// ==================
SetPSTATEFromPSR(bits(N) spsr)
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value
SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32 state
            AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then // AArch32 state
        PSTATE.D  = spsr<27>;
        PSTATE.IT = RestoredITBits(spsr);
        ShouldAdvanceIT = FALSE;
        if HaveDITExt() then PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
        PSTATE.GE = spsr<19:16>;
        PSTATE.E  = spsr<9>;
        PSTATE.<A,I,F> = spsr<8:6>; // No PSTATE.D in AArch32 state
        PSTATE.T  = spsr<5>; // PSTATE.J is RES0
    else // AArch64 state
        if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>;
        PSTATE.<D,A,I,F> = spsr<9:6>; // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;
```

Library pseudocode for shared/functions/system/ShouldAdvanceIT

```c
boolean ShouldAdvanceIT;
```
boolean ShouldAdvanceSS;

SpeculationBarrier();

SynchronizeContext();

SynchronizeErrors();

// Implements the error synchronization event.
SynchronizeErrors();

// Take any pending unmasked physical SError interrupt.
// TakeUnmaskedPhysicalSSErrorInterrupts(boolean iesb_req);

// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSSErrorInterrupts();

bits(32) ThisInstr();

integer ThisInstrLength();

assert FALSE;

boolean UsingAArch32()

// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.
boolean UsingAArch32()

boolean aarch32 = (PSTATE.nRW == '1');
if ![HaveAArch32()] then assert !aarch32;
if ![HaveAArch64()] then assert aarch32;
return aarch32;

boolean VirtualFIQPending();

// Returns TRUE if there is any pending virtual FIQ.
boolean VirtualFIQPending();

boolean VirtualIRQPending();

// Returns TRUE if there is any pending virtual IRQ.
boolean VirtualIRQPending();
Library pseudocode for shared/functions/system/WFxType

enumeration WFxType {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WFIT};

Library pseudocode for shared/functions/system/WaitForEvent

// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wake-up event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
    if !((IsEventRegisterSet() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout))) then
        EnterLowPowerState();
    return;

Library pseudocode for shared/functions/system/WaitForInterrupt

// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wake-up event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

WaitForInterrupt(integer localtimeout)
    if !(HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
Constraint ConstrainUnpredictable(Unpredictable which) 
  case which of 
    when Unpredictable_VMSR    return Constraint_UNDEF; 
    when Unpredictable_WBOVERLAPLD return Constraint_WBSUPPRESS; // return loaded value 
    when Unpredictable_WBOVERLAPST return Constraint_NONE; // store pre-writeback value 
    when Unpredictable_LDPOVERLAP return Constraint_UNDEF; // instruction is UNDEFINED 
    when Unpredictable_BASEOVERLAP return Constraint_UNKNOWN; // use UNKNOWN address 
    when Unpredictable_DATAOVERLAP return Constraint_UNKNOWN; // store UNKNOWN value 
    when Unpredictable_DEVPAGE2 return Constraint_FAULT; // take an alignment fault 
    when Unpredictable_DEVICETAGSTORE return Constraint_NONE; // Do not take a fault 
    when Unpredictable_INSTRDEVICE return Constraint_NONE; // Do not take a fault 
    when Unpredictable_RESCPACR return Constraint_TRUE; // Map to UNKNOWN value 
    when Unpredictable_RESMAIR return Constraint_UNKNOWN; // Map to UNKNOWN value 
    when Unpredictable_SICTAGGED return Constraint_FALSE; // SCTLR_ELx.C == '0' marks address as untagged 
    when Unpredictable_S2RESMEMATTR return Constraint_NC; // Map to Noncacheable value 
    when Unpredictable_RESTEXCR return Constraint_UNKNOWN; // Map to UNKNOWN value 
    when Unpredictable_RESPDACR return Constraint_UNKNOWN; // Map to UNKNOWN value 
    when Unpredictable_RESPRPR return Constraint_UNKNOWN; // Map to UNKNOWN value 
    when Unpredictable_RESVTCRS return Constraint_UNKNOWN; // Map to UNKNOWN value 
    when Unpredictable_RESTnSZ return Constraint_FORCE; // Map to the limit value 
    when Unpredictable_OORTnSZ return Constraint_FORCE; // Map to the limit value 
    when Unpredictable_LARGEIPA return Constraint_FORCE; // Restrict the IA size to the PAMax value 
    when Unpredictable_ESRCONDPASS return Constraint_FALSE; // Report as "AL" 
    when Unpredictable_ILZEROIT return Constraint_FALSE; // Do not zero PSTATE.IT 
    when Unpredictable_ILZEROT return Constraint_FALSE; // Do not zero PSTATE.T 
    when Unpredictable_BPVECTORCATCHPRI return Constraint_TRUE; // Debug Vector Catch: match on 2nd halfword 
    when Unpredictable_VCMATCHHALF return Constraint_FALSE; // No match 
    when Unpredictable_VCMATCHDAPA return Constraint_FALSE; // No match on Data Abort or Prefetch abort 
    when Unpredictable_WPMASKANDRAS return Constraint_FALSE; // Watchpoint disabled 
    when Unpredictable_WPBASCONTOUS return Constraint_FALSE; // Watchpoint disabled 

when Unpredictable RESWPMASK
  return Constraint_DISABLED; // Watchpoint disabled
when Unpredictable WPMAKEDBITS
  return Constraint_FALSE; // Watchpoint disabled
when Unpredictable RESBPWPCCTRL
  return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable BPNOTIMPL
  return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable RESBPWCTYPE
  return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable BPMISMATCHHALF
  return Constraint_FALSE; // No match
when Unpredictable BPMISMATCHHALF
  return Constraint_FALSE; // No match
when Unpredictable RESTARTALIGNPC
  return Constraint_FALSE; // Do not force alignment
when Unpredictable RESTARTZEROUPPERPC
  return Constraint_TRUE; // Do not force alignment
when Unpredictable SMD
  return Constraint_UNDEF; // disabled SMC is Unallocated
when Unpredictable NONFAULT
  return Constraint_FALSE; // Speculation enabled
when Unpredictable SVEZEROUPPER
  return Constraint_TRUE; // top bits of Z registers
when Unpredictable SVELDNFDATA
  return Constraint_TRUE; // Load mem data in NF loads
when Unpredictable SVELDNFZERO
  return Constraint_TRUE; // Write zeros in NF loads
when Unpredictable CHECKSPNONEACTIVE
  return Constraint_TRUE; // Check SP alignment
when Unpredictable NVNV1
  return Constraint_NVNV1_00; // Map unpredictable configuration of HCR_EL2<NV,NV1>
  return Constraint_TRUE; // to NV = 0 and NV1 = 0
when Unpredictable Shareability
  return Constraint_UNDEF; // Map reserved encoding of shareability to outer shareable
when Unpredictable AFUPDATE
  return Constraint_TRUE; // AF update for alignment or permission fault
when Unpredictable IESBinDebug
  return Constraint_TRUE; // Use SCTLR[].IESB in Debug state
when Unpredictable BADPMSFCR
  return Constraint_TRUE; // Bad settings for PMSFCR_EL1/PMSFVEF_EL1/PMSLATFR_EL1
when Unpredictable ZEROBTYPE
  return Constraint_TRUE; // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable CLEARERRITEZERO
  return Constraint_TRUE; // Clearing sticky errors when instruction in flight
when Unpredictable ALUEXCEPTIONRETURN
  return Constraint_FALSE; // Trap to register access in debug state is ignored
when Unpredictable DBGxVR_RESS
  return Constraint_FALSE; // Debug vector registers are not exposed
when Unpredictable PMSCR_PCT
  return Constraint_PMSCR_PCT_VIRT;
when Unpredictable WFXTDEBUG
  return Constraint_FALSE; // WFxT in Debug state does not execute as a NOP
when Unpredictable LS64UNSUPPORTED
  return Constraint_LIMITED_ATOMICITY; // Accesses are not single-copy atomic above the byte level
  // Misaligned exclusives, atomics, acquire/release to region that is not Normal Cacheable WB are atomic
when Unpredictable MISALIGNEDATOMIC
  return Constraint_FALSE; // Trap to register access in debug state is ignored

Shared Pseudocode Functions
when Unpredictable_PMUEVENTCOUNTER
    return Constraint.UNDEF;  // Accesses to the register are UNDEFINED

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONstrained.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint, bits(width)) ConstranUnpredictableBits(Unpredictable which)

    c = ConstranUnpredictable(which);

    if c == Constraint.UNKNOWN then
        return (c, Zeros(width));           // See notes; this is an example implementation only
    elsif c == Constraint.PMSCR_PCT_VIRT then
        return (c, Zeros(width));
    else
        return (c, bits(width) UNKNOWN);    // bits result not used

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

// ConstrainUnpredictableBool()
// ============================

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstranUnpredictable() for more information.

boolean ConstranUnpredictableBool(Unpredictable which)

    c = ConstranUnpredictable(which);
    assert c IN {Constraint.TRUE, Constraint.FALSE};
    return (c == Constraint.TRUE);
// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)
{
    c = ConstrainUnpredictable(which);
    if c == Constraint_UNKNOWN then
        return (c, low);                // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN);    // integer result not used
}

Library pseudocode for shared/functions/unpredictable/Constraint

 enumeration Constraint    {
    General
    Constraint_NONE,                        // Instruction executes with
    Constraint_UNCOND,                     // Instruction executes unconditionally
    Constraint_COND,                       // Instruction executes conditionally
    Constraint_ADDITIONAL_DECODE,          // Instruction executes with additional decode
    // Load-store
    Constraint_WBSUPPRESS,
    Constraint_FAULT,
    Constraint_LIMITED_ATOMICITY,          // Accesses are not single-copy atomic above the
    Constraint_NVNV1_00,
    Constraint_NVNV1_01,
    Constraint_NVNV1_11,
    Constraint_OSH,                        // Constrain to Outer shareable
    Constraint_ISH,                        // Constrain to Inner shareable
    Constraint_NSH,                        // Constrain to Nonshareable
    Constraint_NC,                         // Constrain to Noncacheable
    Constraint_WT,                         // Constrain to Writethrough
    Constraint_WB,                         // Constrain to Writeback
    // IPA too large
    Constraint_FORCE,
    Constraint_FORCENOSLCHECK,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT;
}
enumeration Unpredictable {// VMSR on MVFR
  Unpredictable_VMSR,
  // Writeback/transfer register overlap (load)
  Unpredictable_WBOVERLAPLD,
  // Writeback/transfer register overlap (store)
  Unpredictable_WBOVERLAPST,
  // Load Pair transfer register overlap
  Unpredictable_LDPOVERLAP,
  // Store-exclusive base/status register overlap
  Unpredictable_BASEOVERLAP,
  // Store-exclusive data/status register overlap
  Unpredictable_DATAOVERLAP,
  // Load-store alignment checks
  Unpredictable_DEVPAGE2,
  // Instruction fetch from Device memory
  Unpredictable_INSTRDEVICE,
  // Reserved CPACR value
  Unpredictable_RESCPACR,
  // Reserved MAIR value
  Unpredictable_RESMAIR,
  // Effect of SCTLR_ELx.C on Tagged attribute
  Unpredictable_S1CTAGGED,
  // Reserved Stage 2 MemAttr value
  Unpredictable_S2RESMEMATTR,
  // Reserved TEX:C:B value
  Unpredictable_RESTEXCB,
  // Reserved PRRR value
  Unpredictable_RESPRRR,
  // Reserved DACR field
  Unpredictable_RESDACR,
  // Reserved VTCR.S value
  Unpredictable_RESVTCRS,
  // Reserved TCR.TnSZ value
  Unpredictable_RESTnSZ,
  // Reserved SCTLR_ELx.TCF value
  Unpredictable_RESTCF,
  // Tag stored to Device memory
  Unpredictable_DEVICETAGSTORE,
  // Out-of-range TCR.TnSZ value
  Unpredictable_OORTnSZ,
  // IPA size exceeds PA size
  Unpredictable_LARGEIPA,
  // Syndrome for a known-passing conditional A32 instruction
  Unpredictable_ESRCONDPASS,
  // Illegal State exception: zero PSTATE.IT
  Unpredictable_ILZEROIT,
  // Illegal State exception: zero PSTATE.T
  Unpredictable_ILZEROT,
  // Debug: prioritization of Vector Catch
  Unpredictable_BPVECTORCATCHPRI,
  // Debug Vector Catch: match on 2nd halfword
  Unpredictable_VCMATCHHALF,
  // Debug Vector Catch: match on Data Abort or Prefetch abort
  Unpredictable_VCMATCHDAPA,
  // Debug watchpoints: non-zero MASK and non-ones BAS
  Unpredictable_WPMAKSKANDDBAS,
  // Debug watchpoints: non-contiguous BAS
  Unpredictable_WPBASECONTIGUOUS,
  // Debug watchpoints: reserved MASK
  Unpredictable_RESWPMAK,
  // Debug watchpoints: non-zero MASKed bits of address
  Unpredictable_WPMAKSEDaddy
  // Debug breakpoints and watchpoints: reserved control bits
  Unpredictable_RESBPWPCTRL,
  // Debug breakpoints: not implemented
  Unpredictable_BPNOTIMPL,
  // Debug breakpoints: reserved type
  Unpredictable_RESBPTYPE,
  // Debug breakpoints: not-context-aware breakpoint
  Unpredictable_BPNOTCTXCMP,
Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,

Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,

Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,

Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,

Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,

Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,

Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,

SMC disabled
Unpredictable_SMD,

FF speculation
Unpredictable_NONFAULT,

Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,

Load mem data in NF loads
Unpredictable_SVELDNFDATA,

Write zeros in NF loads
Unpredictable_SVELDNFZERO,

SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,

HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,

Reserved shareability encoding
Unpredictable_Shareability,

Consider Flag Update by HW
Unpredictable_AFUPDATE,

Bad settings for PMSFCR_EL1/PMSEVR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,

Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,

Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,

WFET or WFIT instruction in Debug state
Unpredictable_WFxTDEBUG,

Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,

Misaligned exclusives, atomics, acquire/release to region that is not Normal
Unpredictable_MISALIGNEDATOMIC,

Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,

ALUEXCEPTIONRETURN when in user/system mode in A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,

Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,

Compare DBGVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,

Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,

Reserved PMSCR.PCT behaviour
Unpredictable_PMSCR_PCT,
Library pseudocode for shared/functions/vector/AdvSIMDEndImm

// AdvSIMDEndImm()
// ================

bits(64) AdvSIMDEndImm(bit op, bits(4) cmode, bits(8) imm8)
bits(64) imm64;
case cmode<3:1> of
  when '000'
    imm64 = Replicate(Zeros(24):imm8, 2);
  when '001'
    imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
  when '010'
    imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
  when '011'
    imm64 = Replicate(imm8:Zeros(24), 2);
  when '100'
    imm64 = Replicate(imm8:Zeros(8), 4);
  when '101'
    imm64 = Replicate(imm8:Zeros(8), 4);
  when '110'
    if cmode<0> == '0' then
      imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
    else
      imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
  when '111'
    if cmode<0> == '0' & op == '0' then
      imm64 = Replicate(imm8, 8);
    if cmode<0> == '0' & op == '1' then
      imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
      imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
      imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
      imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
    if cmode<0> == '1' & op == '0' then
      imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
      imm64 = Replicate(imm32, 2);
  if cmode<0> == '1' & op == '1' then
    if UsingAArch32() then ReservedEncoding();
    imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);
return imm64;

Library pseudocode for shared/functions/vector/MatMulAdd

// MatMulAdd()
// ===========
// // Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// // result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
assert N == 128;

bits(N) result;
bits(32) sum;
integer prod;
for i = 0 to 1
  for j = 0 to 1
    sum = Elem[addend, 2*i + j, 32];
    for k = 0 to 7
      prod = Int(Elem[op1, 8*i+k, 8], op1_unsigned) * Int(Elem[op2, 8*j+k, 8], op2_unsigned);
      sum = sum + prod;
    Elem[result, 2*i + j, 32] = sum;
return result;
Library pseudocode for shared/functions/vector/PolynomialMult

```c
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
result = Zeros(M+N);
extended_op2 = ZeroExtend(op2, M+N);
for i=0 to M-1
  if op1<i> == '1' then
    result = result EOR LSL(extended_op2, i);
return result;
```

Library pseudocode for shared/functions/vector/SatQ

```c
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
(result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
return (result, sat);
```

Library pseudocode for shared/functions/vector/SignedSatQ

```c
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)

integer result;
boolean saturated;
if i > 2^(N-1) - 1 then
  result = 2^(N-1) - 1;  saturated = TRUE;
elsif i < -(2^(N-1)) then
  result = -(2^(N-1));  saturated = TRUE;
else
  result = i;  saturated = FALSE;
return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```c
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
assert N == 32;
bits(N) result;
if operand<N-1:N-2> == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
  result = Ones(N);
else
  // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
  // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
  increasedprecision = FALSE;
estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
  // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
  result = estimate<8:0> : Zeros(N-9);
return result;
```
### Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

// UnsignedRecipEstimate()  
// ================       

bits(N) UnsignedRecipEstimate(bits(N) operand)  
assert N == 32;  
bits(N) result;  
if operand<N-1> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF  
    result = Ones(N);  
else // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)  
    // estimate is in the range 256 to 511 representing [1.0 .. 2.0)  
    increasedprecision = FALSE;  
    estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);  
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)  
    result = estimate<8:0> : Zeros(N-9);  
return result;

### Library pseudocode for shared/functions/vector/UnsignedSatQ

// UnsignedSatQ()      
// ==============       

(bits(N), boolean) UnsignedSatQ(integer i, integer N)  
integer result;  
boolean saturated;  
if i > 2^N - 1 then  
    result = 2^N - 1;  saturated = TRUE;  
elsif i < 0 then  
    result = 0;  saturated = TRUE;  
else  
    result = i;  saturated = FALSE;  
return (result<N-1:0>, saturated);

### Library pseudocode for shared/trace/Common/GetTimestamp

// GetTimestamp()       
// ==============       
// Returns the Timestamp depending on the type  

bits(64) GetTimestamp(TimeStamp timeStampType)  
case timeStampType of  
    when TimeStamp_Physical  
        return PhysicalCountInt();  
    when TimeStamp_Virtual  
        return PhysicalCountInt() - CNTVOFF_EL2;  
    when TimeStamp_OffsetPhysical  
        return PhysicalCountInt() - CNTPOFF_EL2;  
    when TimeStamp_None  
        return Zeros(64);  
    when TimeStamp_CoreSight  
        return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";  
    otherwise  
        Unreachable();  

### Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

// EffectiveE0HTRE()       
// ===============         
// Returns effective E0HTRE value  

ten EffectiveE0HTRE()  
    return if ELUSingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;

Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;

Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;

Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    if ! (HaveTraceExt() && HaveSelfHostedTrace()) then return FALSE;
    if EDS.CR.TFO == '0' then return TRUE;
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
        if secure_trace_enable == '1' && ! ExternalSecureNoninvasiveDebugEnabled() then return TRUE;
    else
        if SecureOnlyImplementation() && ! ExternalSecureNoninvasiveDebugEnabled() then return TRUE;
    return FALSE;
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
if !HaveTraceExt() then return FALSE;
ss = SecurityStateAtEL(el);
if SelfHostedTraceEnabled() then
boolean trace_allowed;
// Detect scenarios where tracing in this Security state is never allowed.
case ss of
    when SS_NonSecure
        trace_allowed = TRUE;
    when SS_Secure
        bit trace_bit;
        if HaveEL(EL3) then
            trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
        else
            trace_bit = '1';
        trace_allowed = trace_bit == '1';
    bit TRE_bit;
    case el of
        when EL3 TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
        when EL2 TRE_bit = EffectiveE2TRE();
        when EL1 TRE_bit = EffectiveE1TRE();
        when EL0
            if EffectiveTGE() == '1' then
                TRE_bit = EffectiveE0HTRE();
            else
                TRE_bit = EffectiveE0TRE();
    return trace_allowed && TRE_bit == '1';
else
    case ss of
        when SS_NonSecure return ExternalNoninvasiveDebugEnabled();
        when SS_Secure return ExternalSecureNoninvasiveDebugEnabled();

// TraceContextIDR2()
// ==================

boolean TraceContextIDR2()
if !TraceAllowed(PSTATE.EL) || !HaveEL(EL2) then return FALSE;
return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');

// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers

TraceSynchronizationBarrier();
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
if SelfHostedTraceEnabled() then
    if HaveEL(EL2) then
        TS_el2 = TRFCR_EL2.TS;
        if !HaveECVExt() && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);

        case TS_el2 of
            when '00'
                // Falls out to check TRFCR_EL1.TS
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable(); // ConstrainUnpredictableBits removes this case
        endcase
    else
        return TimeStamp_CoreSight;
    endif
end

Library pseudocode for shared/translation/at/ATAccess

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
Library pseudocode for shared/translation/at/EncodePARAttrs

// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;
    if HaveMTEExt() && memattrs.tagged then
        result<7:0> = '11110000';
        return result;
    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        if memattrs.device == DeviceType_nGnRnE then
            result<3:0> = '0000';
            return result;
        elseif memattrs.device == DeviceType_nGnRE then
            result<3:0> = '0100';
        elseif memattrs.device == DeviceType_nGRE then
            result<3:0> = '1000';
        else // DeviceType_GRE
            result<3:0> = '1100';
    else
        if memattrs.outer.attrs == MemAttr_WT then
            result<7:6> = if memattrs.outer.transient then '00' else '10';
            result<5:4> = memattrs.outer.hints;
        elseif memattrs.outer.attrs == MemAttr_WB then
            result<7:6> = if memattrs.outer.transient then '01' else '11';
            result<5:4> = memattrs.outer.hints;
        else // MemAttr_NC
            result<7:4> = '0100';
        if memattrs.inner.attrs == MemAttr_WT then
            result<3:2> = if memattrs.inner.transient then '00' else '10';
            result<1:0> = memattrs.inner.hints;
        elseif memattrs.inner.attrs == MemAttr_WB then
            result<3:2> = if memattrs.inner.transient then '01' else '11';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
            result<3:0> = '0100';
    return result;

Library pseudocode for shared/translation/at/PAREncodeShareability

// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
        memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';
    case memattrs.shareability of
        when Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';

Library pseudocode for shared/translation/at/TranslationStage

enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};
Library pseudocode for shared/translation/attrs/DecodeDevice

// DecodeDevice()
// ==============
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00' return DeviceType_nGnRnE;
        when '01' return DeviceType_nGnRE;
        when '10' return DeviceType_nGRE;
        when '11' return DeviceType_GRE;

Library pseudocode for shared/translation/attrs/DecodeLDFAttr

// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    if attr == 'x0xx' then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldfattr.attrs = MemAttr_WB; // Write-back
    else Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate
        endcase;

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC & ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';
    endcase;

    return ldfattr;

Library pseudocode for shared/translation/attrs/DecodeSDFAttr

// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    case rgn of
        when '00'               // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01'               // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10'               // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11'               // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;
        endcase;

    sdfattr.transient = FALSE;

    return sdfattr;
Library pseudocode for shared/translation/attrs/DecodeShareability

// DecodeShareability()
// ===============
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability_OSH;
        when '11' return Shareability_ISH;
        when '00' return Shareability_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH return Shareability_OSH;
                when Constraint_ISH return Shareability_ISH;
                when Constraint_NSH return Shareability_NSH;

Library pseudocode for shared/translation/attrs/EffectiveShareability

// EffectiveShareability()
// ==============
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
         memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;

Library pseudocode for shared/translation/attrs/MAIRAttr

// MAIRAttr()
// =========
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) MAIRAttr(integer index, MAIRType mair)
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;

Library pseudocode for shared/translation/attrs/NormalNCMemAttr

// NormalNCMemAttr()
// ================
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;

    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype = MemType_Normal;
    nc_memattrs.outer = non_cacheable;
    nc_memattrs.inner = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;
    nc_memattrs.tagged = FALSE;

    return nc_memattrs;
boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean s1aarch64)
    case attr of
        when '0000xx01' return !(s1aarch64 && HaveFeatXS());
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(s1aarch64 && HaveFeatXS());
        when '10100000' return !(s1aarch64 && HaveFeatXS());
        when '11110000' return !(s1aarch64 && HaveMTE2Ext());
        when 'xxxx0000' return TRUE;
        otherwise return FALSE;
Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

// S1DecodeMemAttrs()
// ================
// Decode MAIR-format memory attributes assigned in stage 1
MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    MemoryAttributes memattrs;
    case attr of
        when '0000xxxx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device = DecodeDevice(attr<3:2>);
            memattrs.tagged = FALSE;
            memattrs.xs = if s1aarch64 then NOT attr<0> else '1';
        when '01000000'
            assert s1aarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged = FALSE;
            memattrs.outer.attrs = MemAttr_NC;
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.xs = '0';
        when '10100000'
            assert s1aarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged = FALSE;
            memattrs.outer.attrs = MemAttr_WT;
            memattrs.outer.hints = MemHint_RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs = MemAttr_WT;
            memattrs.inner.hints = MemAttr_RA;
            memattrs.xs = '0';
        when '11110000' // Tagged memory
            assert s1aarch64 && HaveMTE2Ext();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged = TRUE;
            memattrs.outer.attrs = MemAttr_WB;
            memattrs.outer.hints = MemHint_RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs = MemAttr_WB;
            memattrs.inner.hints = MemHint_RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs = '0';
        otherwise
            memattrs.memtype = MemType_Normal;
            memattrs.outer = DecodeLDFAttr(attr<7:4>);
            memattrs.inner = DecodeLDFAttr(attr<3:0>);
            memattrs.tagged = FALSE;
            if (memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';
    memattrs.shareability = DecodeShareability(sh);
    return memattrs;
### Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```c
// S2CombineS1AttrHints()
// ================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
{
    MemAttrHints attrhints;
    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;
    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints     = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;
    return attrhints;
}
```

### Library pseudocode for shared/translation/attrs/S2CombineS1Device

```c
// S2CombineS1Device()
// ===============
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
{
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
}
```
Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

// S2CombineS1MemAttrs()
// =====================
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(
    MemoryAttributes s1_memattrs,
    MemoryAttributes s2_memattrs)

    MemoryAttributes memattrs;
    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else                                                // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner  = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer   = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);
    if ELUsingAArch32(EL2) || !HaveMTE2Ext() then
        memattrs.tagged = FALSE;
    else
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                        s2_memattrs.shareability);
    memattrs.xs           = s2_memattrs.xs;
    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;

Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(
    Shareability s1_shareability,
    Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH || s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH || s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
Library pseudocode for shared/translation/attrs/S2DecodeCacheability

// S2DecodeCacheability()
// ===============
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;
    case attr of
        when '01' s2attr.attrs = MemAttr_NC; // Non-cacheable
        when '10' s2attr.attrs = MemAttr_WT; // Write-through
        when '11' s2attr.attrs = MemAttr_WB; // Write-back
        otherwise // Constrained unpredictable
            case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
                when Constraint_NC s2attr.attrs = MemAttr_NC;
                when Constraint_WT s2attr.attrs = MemAttr_WT;
                when Constraint_WB s2attr.attrs = MemAttr_WB;
    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints = bits(2) UNKNOWN;
s2attr.transient = boolean UNKNOWN;
    return s2attr;

Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

// S2DecodeMemAttrs()
// ================
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh)
    MemoryAttributes memattrs;
    case attr of
        when '00xx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device = DecodeDevice(attr<1:0>);
        otherwise // Normal memory
            memattrs.memtype = MemType_Normal;
            memattrs.outer = S2DecodeCacheability(attr<3:2>);
            memattrs.inner = S2DecodeCacheability(attr<1:0>);
        memattrs.shareability = DecodeShareability(sh);
    return memattrs;
Library pseudocode for shared/translation/attrs/WalkMemAttrs

```plaintext
// WalkMemAttrs()
// =============
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
    walkmemattrs.outer        = DecodeSDFAttr(orgn);
    walkmemattrs.tagged       = FALSE;

    if (walkmemattrs.inner.attrs == MemAttr_WB &&
        walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';

    return walkmemattrs;
```

Library pseudocode for shared/translation/faults/AlignmentFault

```plaintext
// AlignmentFault()
// ===============

FaultRecord AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)
    FaultRecord fault;

    fault.statuscode  = Fault_Alignment;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.secondstage = secondstage;

    return fault;
```

Library pseudocode for shared/translation/faults/AsyncExternalAbort

```plaintext
// AsyncExternalAbort()
// ==============
// Return a fault record indicating an asynchronous external abort

FaultRecord AsyncExternalAbort(boolean parity, bits(2) errortype, bit extflag)
    FaultRecord fault;

    fault.statuscode = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    fault.extflag   = extflag;
    fault.errortype = errortype;
    fault.acctype   = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;

    return fault;
```
Library pseudocode for shared/translation/faults/NoFault

// NoFault()
// =========
// Return a clear fault record indicating no faults have occurred

FaultRecord NoFault()
{    
    FaultRecord fault;
    fault.statuscode = Fault_None;
    fault.acctype = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;
    return fault;
}

Library pseudocode for shared/translation/translation/S1TranslationRegime

// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
{    
    if el != FL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;
    }

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
{    
    return S1TranslationRegime(PSTATE.EL);
}

Library pseudocode for shared/translation/vmsa/AddressDescriptor

type AddressDescriptor is (    
    FaultRecord fault,       // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress paddress,
    bits(64) vaddress
)

custom integer FINAL_LEVEL = 3;
Library pseudocode for shared/translation/vmsa/ContiguousSize

```c
// ContiguousSize()
// ================
// Return the number of entries log2 marking a contiguous output range

integer ContiguousSize(TGx tgx, integer level)
    case tgx of
        when TGx_4KB
            assert level IN {1, 2, 3};
            return 4;
        when TGx_16KB
            assert level IN {2, 3};
            return if level == 2 then 5 else 7;
        when TGx_64KB
            assert level IN {2, 3};
            return 5;
```

Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```c
// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa, MemoryAttributes memattrs)
    AddressDescriptor addrdesc;
    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault();
    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```c
// CreateFaultyAddressDescriptor()
// ===============================
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;
    addrdesc.vaddress = va;
    addrdesc.fault    = fault;
    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/DescriptorType

```c
enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Block,
    DescriptorType_Page,
    DescriptorType_Invalid
};
```

Library pseudocode for shared/translation/vmsa/Domains

```c
constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```
Library pseudocode for shared/translation/vmsa/FetchDescriptor

// FetchDescriptor()
// ===============
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress, FaultRecord fault_in)
// 32-bit descriptors for AArch32 Short-descriptor format
// 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
assert N == 32 || N == 64;
bits(N) descriptor;
FaultRecord fault = fault_in;
AccessDescriptor walkacc;

walkacc.acctype = AccType_TTW;
// MPAM PARTID for translation table walk is determined by the access invoking the translation
walkacc.mpam = GenMPAMcurEL(fault.acctype);

PhysMemRetStatus memstatus;
(memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
if IsFault(memstatus) then
  fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress, walkacc, N DIV 8, fault);
  if IsFault(fault.statuscode) then
    return (fault, bits(N) UNKNOWN);
else ee == '1' then
  descriptor = BigEndianReverse(descriptor);
return (fault, descriptor);

Library pseudocode for shared/translation/vmsa/HasUnprivileged

// HasUnprivileged()
// ===============
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
  return (regime IN {
    Regime_EL20,
    Regime_EL30,
    Regime_EL10
  });

Library pseudocode for shared/translation/vmsa/IsAtomicRW

// IsAtomicRW()
// ===========
// Is the access an atomic operation?

boolean IsAtomicRW(AccType acctype)
  return acctype IN {
    AccType_ATOMICRW,
    AccType_ORDEREDRW,
    AccType_ORDEREDATOMICRW
  }

Library pseudocode for shared/translation/vmsa/Regime

enumeration Regime {
  Regime_EL3,       // EL3
  Regime_EL30,      // EL3&0 (PL1&0 when EL3 is AArch32)
  Regime_EL2,       // EL2
  Regime_EL20,      // EL2&0
  Regime_EL10       // EL1&0
};
Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

// RegimeUsingAArch32()
// ====================================
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
{
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2 return ELUsingAArch32(EL2);
        when Regime_EL3 return FALSE;
}

Library pseudocode for shared/translation/vmsa/S1TTWParams

type S1TTWParams is (  
    // A64-VMSA exclusive parameters
    bit ha, // TCR_ELx.HA
    bit hd, // TCR_ELx.HD
    bit tbi, // TCR_ELx.TBI{x}
    bit tbid, // TCR_ELx.TBID{x}
    bit nfd, // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit e0pd, // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit ds, // TCR_ELx.DS
    bits(6) tsxs, // TCR_ELx.TxSZ
    bit epan, // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit dct, // HCR_EL2.DCT
    bit nvl, // HCR_EL2.NV1
    bit cmow, // SCTLR_EL1.CMW0 or SCTLR_EL2.CMW0 when HCR_EL2.E2H == '1'
    // A32-VMSA exclusive parameters
    bits(3) t0sz, // TTBCR.T0SZ
    bits(3) t1sz, // TTBCR.T1SZ
    bit uwxn, // SCTLR.UWXN
    // Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx txg, // TCR_ELx.TGx / Always TGx_4KB
    bits(2) irgn, // TCR_ELx.IRGNx / TTBCR.IRGNx or HTCR.IRGNO
    bits(2) orgn, // TCR_ELx.ORGNx / TTBCR.ORGNx or HTCR.ORGNO
    bits(2) sh, // TCR_ELx.SHx / TTBCR.SHx or HTCR.SH0
    bit hpd, // TCR_ELx.HPD{x} / TTBCR2.HPDx or HTCR.HPD
    bit ee, // SCTLR_ELx.EE / SCTLR.EE or HSCTRL.EE
    bit wxn, // SCTLR_ELx.WXN / SCTLR.WXN or HTCLR.WXN
    bit ntsmd, // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSTCTRL.nTLSMD
    bit dc, // HCR_EL2.DC / HCR.DC
    bit sif, // SCR_EL3.SIF / SCR.SIF
    MAIRType mair // MAIR_ELx / MAIR:MAIR0 or HMAIR1:HMAIR0
)
Library pseudocode for shared/translation/vmsa/S2TTWParams

type S2TTWParams is (  
    // A64-VMSA exclusive parameters  
    bit ha,     // VTCR_EL2.HA  
    bit hd,     // VTCR_EL2.HD  
    bit sl2,    // V{S}TCR_EL2.SL2  
    bit ds,     // VTCR_EL2.DS  
    bit sw,     // VSTCR_EL2.SW  
    bit ns,     // V{S}TCR_EL2.SW  
    bit sa,     // VSTCR_EL2.SA  
    bit nsa,    // VTCR_EL2.NSA  
    bits(3) ps, // VTCR_EL2.PS  
    bits(6) txsz,  // V{S}TCR_EL2.T0SZ  
    bit fwb,    // HCR_EL2.PTW  
    bit cmow,   // HCRX_EL2.CMOW  
    // A32-VMSA exclusive parameters  
    bit s,      // VTCR.S  
    bits(4) t0sz,  // VTCR.T0SZ  
    // Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)  
    TGx         tgx,  // V{T}CR_EL2.TG0  / Always TGx_4KB  
    bits(2) sl0, // V{T}CR_EL2.SL0  / VTCR.SL0  
    bits(2) irgn, // V{T}CR_EL2.IRGN0  / VTCR.IRGN0  
    bits(2) orgn, // V{T}CR_EL2.ORGN0  / VTCR.ORGN0  
    bits(2) sh,  // VTCR_EL2.SH0  / VTCR.SH0  
    bit ee,     // SCTLR_EL2.EE  / HSCTRL.EE  
    bit ptw,    // HCR_EL2.PTW  / HCR.PTW  
    bit vm,     // HCR_EL2.VM  / HCR.VM  
  )

Library pseudocode for shared/translation/vmsa/SDFType

enumeration SDFType {  
    SDFType_Table,  
    SDFType_Invalid,  
    SDFType_Supersection,  
    SDFType_Section,  
    SDFType_LargePage,  
    SDFType_SmallPage  
};

Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

// SecurityStateForRegime()  
// =========================  
// Return the Security State of the given translation regime  
SecurityState SecurityStateForRegime(Regime regime)
  case regime of  
    when Regime_EL3 return SecurityStateAtEL(EL3);  
    when Regime_EL30 return SS_Secure; // A32 EL3 is always Secure  
    when Regime_EL2 return SecurityStateAtEL(EL2);  
    when Regime_EL20 return SecurityStateAtEL(EL2);  
    when Regime_EL10 return SecurityStateAtEL(EL1);  
  )
Library pseudocode for shared/translation/vmsa/StageOA

// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, TGx tgx, TTWState walkstate)
// Output Address
FullAddress oa;

integer csize;
tsize = TranslationSize(tgx, walkstate.level);
if walkstate.contiguous == '1' then
  csize = ContiguousSize(tgx, walkstate.level);
else
  csize = 0;

ia_msb = tsize + csize;
oa.paspace = walkstate.baseaddress.paspace;
oa.address = walkstate.baseaddress.address<51:ia_msb>:ia<ia_msb-1:0>;

return oa;

Library pseudocode for shared/translation/vmsa/TGx

enumeration TGx {
  TGx_4KB,
  TGx_16KB,
  TGx_64KB
};

Library pseudocode for shared/translation/vmsa/TGxGranuleBits

// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
  case tgx of
    when TGx_4KB return 12;
    when TGx_16KB return 14;
    when TGx_64KB return 16;

Library pseudocode for shared/translation/vmsa/TLBContext

type TLBContext is (SecurityState ss,
  Regime regime,
  bits(16) vmid,
  bits(16) asid,
  bit nG,
  PASpace ipaspace, // Used in stage 2 lookups & invalidations only
  boolean includes_s1,
  boolean includes_s2,
  bits(64) ia,       // Input Address
  TGx tg,
  bit cnp,
  bit xs            // XS attribute (FEAT_XS)
)
Library pseudocode for shared/translation/vmsa/TLBRecord

type TLBRecord is |
| TLBContext context, |
| TTWState walkstate, |
| integer blocksize, // Number of bits directly mapped from IA to OA |
| integer contigsize, // Number of entries log 2 marking a contiguous output range |
| bits(64) sidescriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1) |
| bits(64) s2descriptor // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2) |

Library pseudocode for shared/translation/vmsa/TTWState

type TTWState is |
| boolean istable, |
| integer level, |
| FullAddress baseaddress, |
| bit contiguous, |
| bit nG, |
| bit guardedpage, |
| SDFType sdftype, // AArch32 Short-descriptor format walk only |
| bits(4) domain, // AArch32 Short-descriptor format walk only |
| MemoryAttributes memattrs, |
| Permissions permissions |

Library pseudocode for shared/translation/vmsa/TranslationRegime

// TranslationRegime()
// ===============
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el, AccType acctype)
  if el == EL3 then
    return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
  elsif el == EL2 then
    return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
  elsif el == EL1 then
    if acctype == AccType_NV2REGISTER then
      assert EL2Enabled();
      return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    else
      return Regime_EL10;
  elsif el == EL0 then
    if IsSecure() & ELUsingAArch32(EL3) then
      return Regime_EL30;
    elsif ELIsInHost(EL0) then
      return Regime_EL20;
    else
      return Regime_EL10;
  else
    Unreachable();

Library pseudocode for shared/translation/vmsa/TranslationSize

// TranslationSize()
// ================
// Compute the number of bits directly mapped from the input address
// to the output address

integer TranslationSize(TGx tgx, integer level)
  granulebits = TGxGranuleBits(tgx);
  blockbits = (FINAL_LEVEL - level) * (granulebits - 3);
  return granulebits + blockbits;
**Library pseudocode for shared/translation/vmsa/UseASID**

```java
// UseASID()
// =========
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext access)
    return HasUnprivileged(access.regime);
```

**Library pseudocode for shared/translation/vmsa/UseVMID**

```java
// UseVMID()
// =========
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(TLBContext access)
    return access.regime == Regime_EL10 && EL2Enabled();
```

**Library pseudocode for shared/translation/vmsa/VARange**

```java
enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

Internal version only: isa v33.16decrel, AdvSIMD v29.05, pseudocode v2021-12_rel, sve v2021-12 ; Build timestamp: 2021-12-15T12:33

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.